Non-linear Data Structure

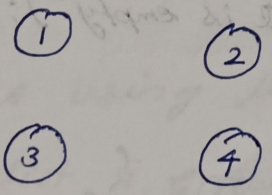
1) Graph
2) Tree

$G(V, E)$
Graph collection of V & E
V: set of vertices
E: set of edges

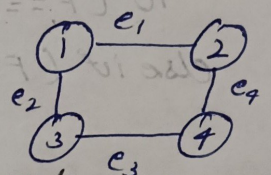
V = non-empty होगा always

E = Empty हो सकता है।

ऐसा graph जिसमें कोई edge ना present हो उसे NULL Graph बोलते है.



NULL Graph



undirected Graph

$V = \{1, 2, 3, 4\}$

$E = \{e_1, e_2, e_3, e_4\}$

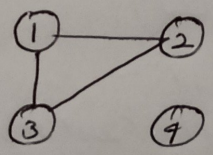
$e_1 = (1,2)$ or $(2,1)$

$e_2 = (1,3)$ or $(3,1)$

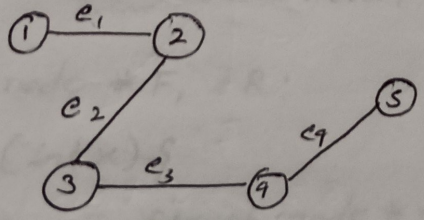
$V = \{1, 2, 3, 4\}$

(3) ⇒ इसको भी graph बोल सकते है।

Tree :- A tree is a connected acyclic graph अगर vertex दिख गया मतलब Graph है।



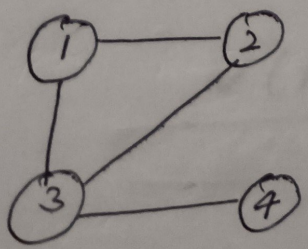
Disconnected Graph



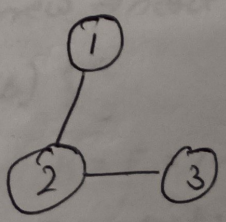
Tree

T is subset of Graph

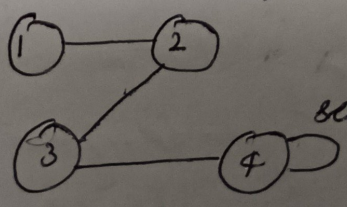
**Self loop भी एक तरह का cycle होता है।



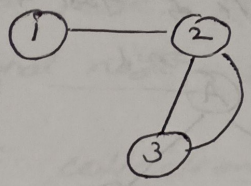
Graph but not tree



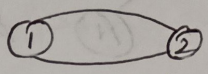
Tree & Graph both



Graph but not tree



Graph but not Tree



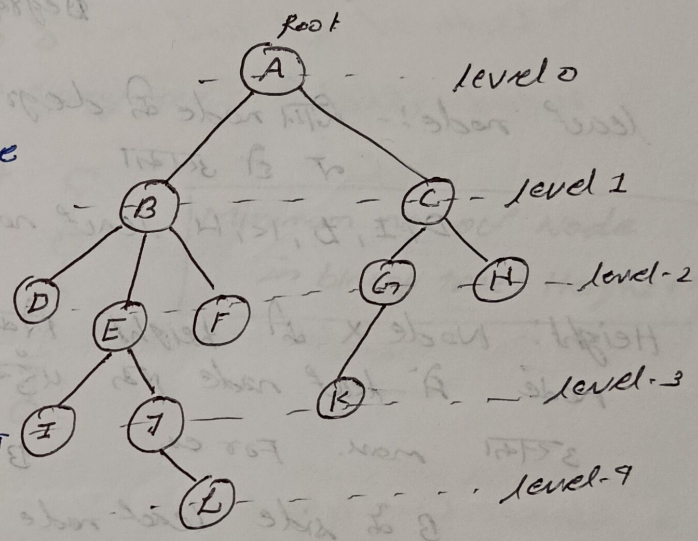
parallel edge

* अगर कोई ग्राफ Graph Tree है तो उसमें (V-1) edges होंगे.

→ Every Tree is Graph. True
→ Every Graph is Tree. False

→ Tree (Acyclic connected Graph is known as Tree)

Tree is a non-linear hierarchial data structure



Root :- पूरे tree में एक node ही root होता है जहाँ से Tree start हुआ

Root node - A

Parent : Just उस node के ऊपर level वाला node Parent node होगा जिससे वो derive हुआ है.

E का parent node B होगा
J का parent node E होगा।
H का parent node C होगा।

Child : parent node से जो derive node है.

E का child है I & J
J का child है L
D, E, F child है A का

Sibling : वो सब node जिसके parents एक ही हो.

D, E, F sibling हैं।

Ancestor :- Node X का Ancestor पता करना है तो Root node से X तक जाओ रास्ते में जो node मिले वो सब ancestor होते है.

L का ancestor B, E, J

I का ancestor B, E

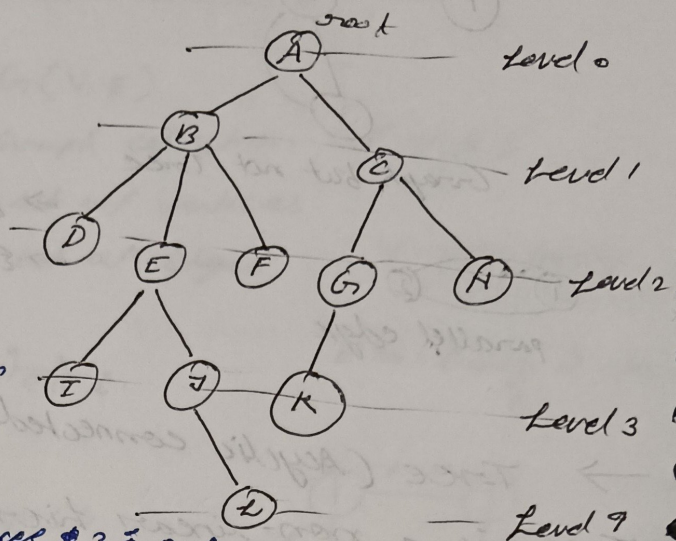
A → B → E → J → L

ancestor A → B → E → I

Descendent :- उस node के level के नीचे जितने node है सब descendent है B का descendent node है D, E, F, G, H, I, J, K, L

Degree:- उस node के कितने children उस node के just नीचे वाले level में

E का degree = 2



* Tree की degree निकालना है तो पूरे tree में कितने node है उन सब की individual degree निकाल जो degree maximum होगी वही tree का degree होगा। यहि Tree का degree = 3

Degree = 3 है B की max.

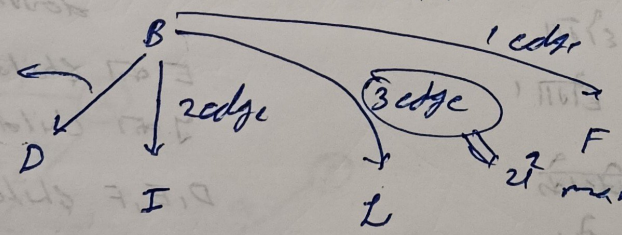
leaf node:- जिस node की degree zero मतलब कोई children न हो उसका

D, I, L, K, H leaf nodes हैं।

Height:- Node X की height निकालना है तो उसके descendent node में leaf node तक पहुँचने पर जो edge आ रहा उसका num. For eg.

B की height calculate करना है तो B के side leaf node है

D, I, L, F



ये max है तो 3

Height of B = 3 होगी निकलते हैं।

Tree की height = Root node की height

Depth:- Root node से वहाँ तक पहुचने में कितने edge participate करेंगे

J की depth है 3

Depth of tree: सबसे नीचे level पर जो ~~root~~ leaf node है उसकी depth.

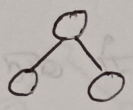
Interval node:- जिस node के पास atleast one children है वो Interval node. Leaf node को छोड़कर सब interval node है.

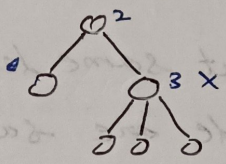
Forest:- collection of tree, जो आपस में connected नहीं है

Binary Tree

↳ At most 2 children (0, 1, 2)

Binary Tree उस Tree को बोलेंगे जिसके सारे node के children atmost 2 हो।



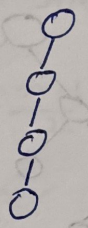


Not a Binary tree

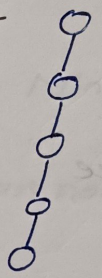
Binary tree

h=2

h = 3

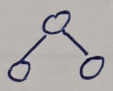


h=4

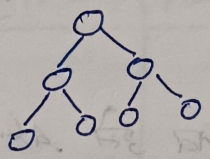


Minimum no. of Node in binary tree = Height + 1

h=1

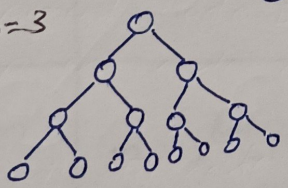


h=2



Maximum no. of Node in binary tree: $2^{Height+1} - 1$

h=3



1+2+4+8

So, maximum node के लिए G.P बनेगा

$$2^0 + 2^1 + 2^2 + \cdots 2^n = \frac{1 \cdot (2^n - 1)}{2-1} = 2^n - 1$$

$$Height + 1 \le n \le 2^{Height+1} - 1$$

↓ No. of node in binary tree

* Height varies as follows

$$\log_2(n+1) - 1 \le h \le n - 1$$

↓↓

Maximum height किसी भी binary tree: $O(n)$

Minimum height " " " $O(\log_2 n)$

*

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

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

$h + 1 = \log_2(n+1)$

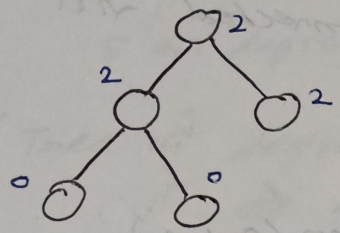
$h = \log_2(n+1) - 1$

$n = h + 1$

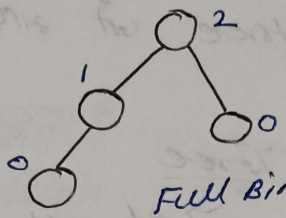
$h = n - 1$

* Full Binary Tree / strict Binary Tree

Tree जिसके या तो 0 children हो या 2 children



Full Binary Tree

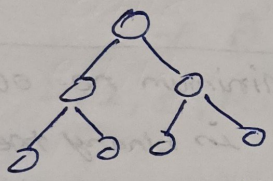


Full Binary Tree नहीं है।

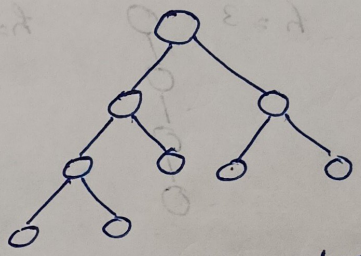
* Complete Binary Tree

→ All leaf nodes are at same level.
→ Depth of all leaf node are same.
> दोनों बात एक है।

H तक level filled है maximum node है इस binary Tree



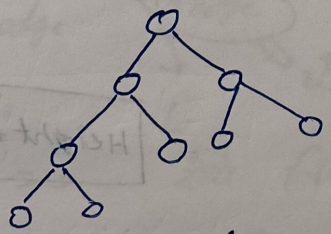
Complete binary Tree



Full Binary tree है but not complete binary tree

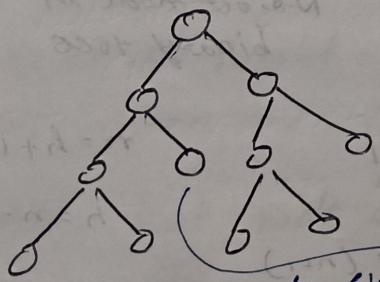
* Almost Complete Binary Tree

left to right fill कर रहे और last level fill नहीं है तब इसे Almost complete Binary Tree बोलेंगे।



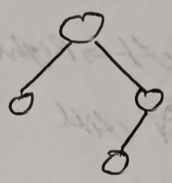
⇒ Full नहीं है

Almost complete Binary Tree



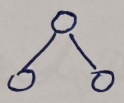
Left to right fill करने level wise तब → में नहीं node पहले fill होना चाहिए ∴ ये Almost complete Binary tree नहीं है।

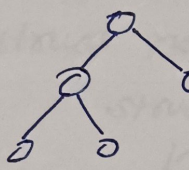
* All complete binary tree are Full binary tree but all Full binary tree are not always complete binary tree.



⇒ ये किसी binary tree है
अगर ये ना तो Full binary tree है, ना तो complete binary tree है, ना तो Almost complete binary tree.

Full Binary tree में min node

h=1  3

h=2  5

$n = 2h+1$ से आएगा Max. node तो $2^{h+1}-1$ ही होगा।

Max. height = $\frac{n-1}{2}$ Min height = $\log(n+1)-1$

Complete Binary tree में max node = min. node = $2^{h+1}-1$

	Binary Tree	Full Binary Tree	Complete Binary Tree
minimum node	$n = h+1$	$n = 2h+1$	$n = 2^{h+1}-1$
maximum node	$n = 2^{h+1}-1$	$n = 2^{h+1}-1$	$n = 2^{h+1}-1$
minimum height	$h = \log_2(n+1)-1$	$h = \log_2(n+1)-1$	$h = \log_2(n+1)-1$
maximum height	$h = n-1$	$h = \frac{n-1}{2}$	$h = \log_2(n+1)-1$

For Complete Binary Tree

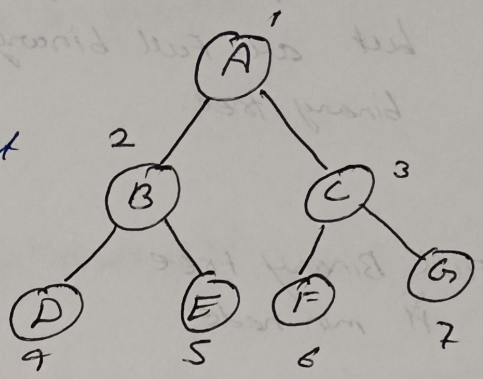
Number of leaf node = 2^h

Number of internal node = 2^h-1

Binary Tree Representation

→ Array representation

Tree को level wise Left → Right traverse करेंगे और Array में add कर देंगे।



	A	B	C	D	E	F	G
	1	2	3	4	5	6	7

✗ ⇓ 1 से भी start हो सकता था।

For particular index i

LChild = 2i ⟹ इसका मतलब है leftChildren 2i th index मिलेगा

RChild = 2i + 1

Parent = $[i/2]$ → floor

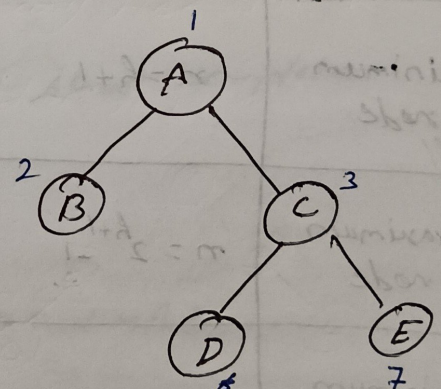
D का left children
i = 4
2i = 8 present ही नहीं है

C का left children
i = 3
2i = 6

F का parent
i = 6

$[6/2] = 3$ → C

E का parent = $[5/2] = 2$ → B



	A	B	C	N	N	D	E
	1	2	3	4	5	6	7

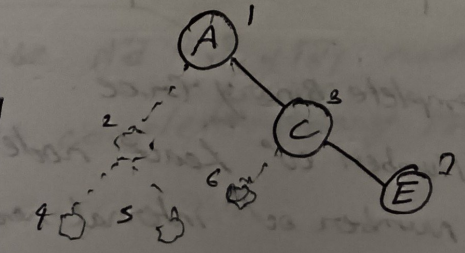
NULL, NULL

NULL लिख देंगे अगर present नहीं है तो Child

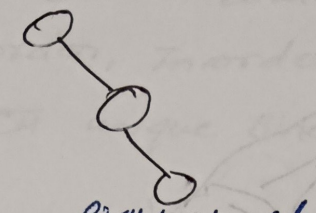
So, ये representation complete Binary tree के लिए सही और बाकी के लिए memory का wastage हो जाएगा।

→ Almost complete BT

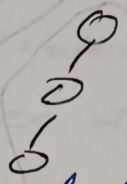
	A	N	C	N	N	N	E
	1	2	3	4	5	6	7



Array representation heap में use होता है।



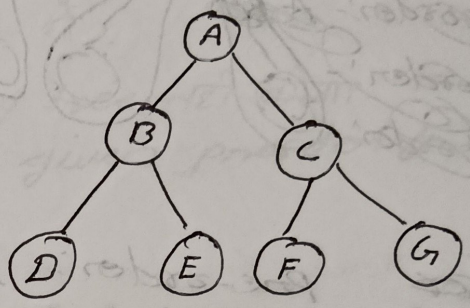
Right skewed Binary Tree

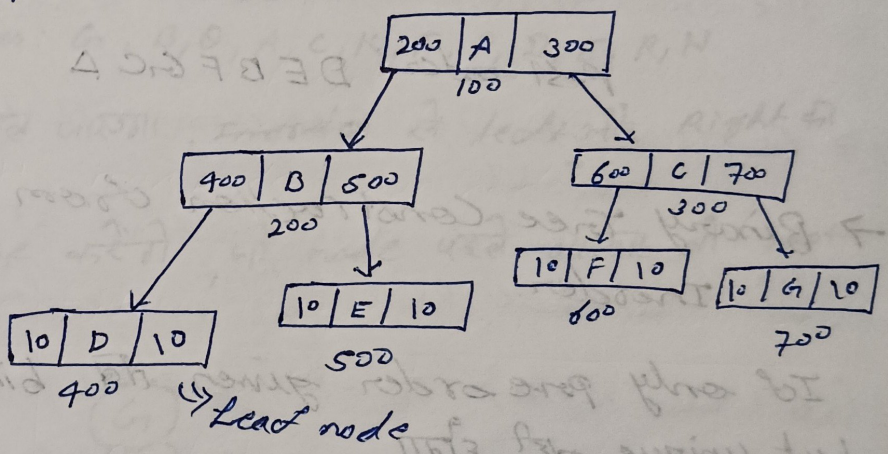


Left skewed binary Tree

→ Linked list Representation

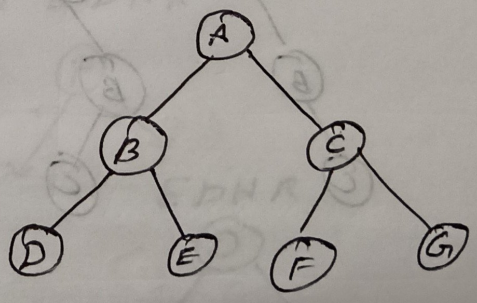
```
struct node{
    struct node *lp;
    int data;
    struct node *rp;
};
```





→ Traversal

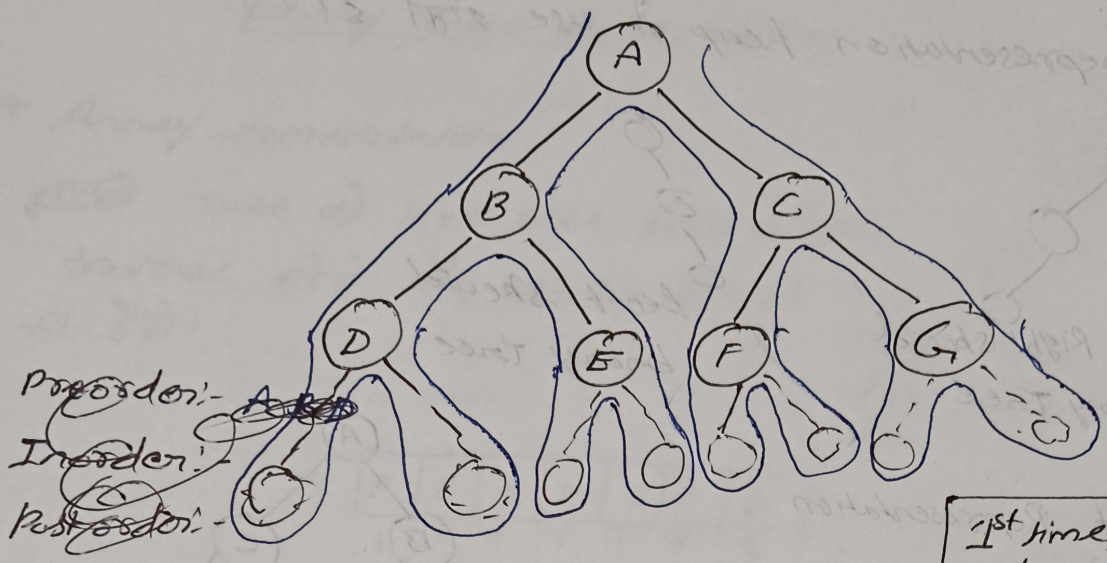
Preorder: Root, Left, Right
Inorder: Left, Root, Right
Post-order: Left, Right, Root



कैसे निकालते हैं सबका।

Dummy children बना देंगे अगर present नहीं है children तब

Traversal करेंगे जो 1st time मिलेगा preorder में लिख देंगे जो 2nd time मिलेगा Inorder में 3rd time मिलेगा Postorder में।



1st time - pre order
2nd time - Inorder
3rd time - Post order

Pre-order : A B D E C F G

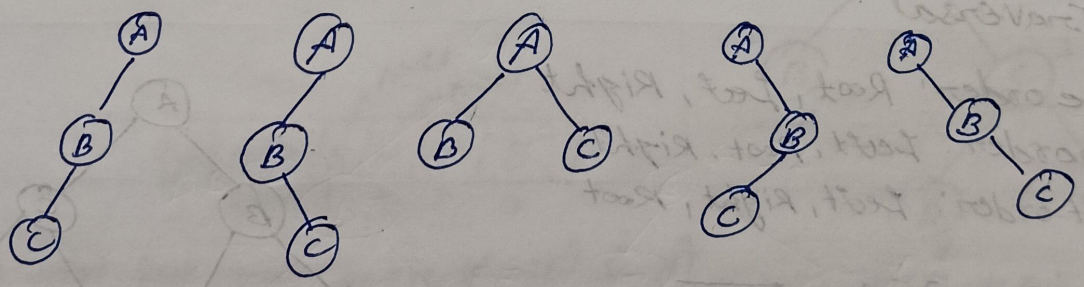
Inorder : D B E A F C G

Post-order : D E B F G C A

→ Binary Tree Construction from Preorder, postorder, Inorder.

If only pre order given तब binary tree बना तो लेंगे but unique नहीं होगा

E.g Preo-order :- A, B, C



* If any one order is given मतलब Inorder, postorder, preorder में से कोई एक भी दिया हो तब

$$\text{No. of possible Binary Tree} = \frac{2^nC_n}{n+1}$$

n = no. of node in given traversal

* If only one traversal is given then we are not not able to draw unique Binary tree
Pre-order, Inorder, Post-order में सिर्फ कोई एक ही दिया है तो हम unique Binary tree नहीं बना पाएंगे।

* unique tree नहीं बनेगा जब **Inorder + pre-order** दिया हो या **Inorder + post-order** दिया है।

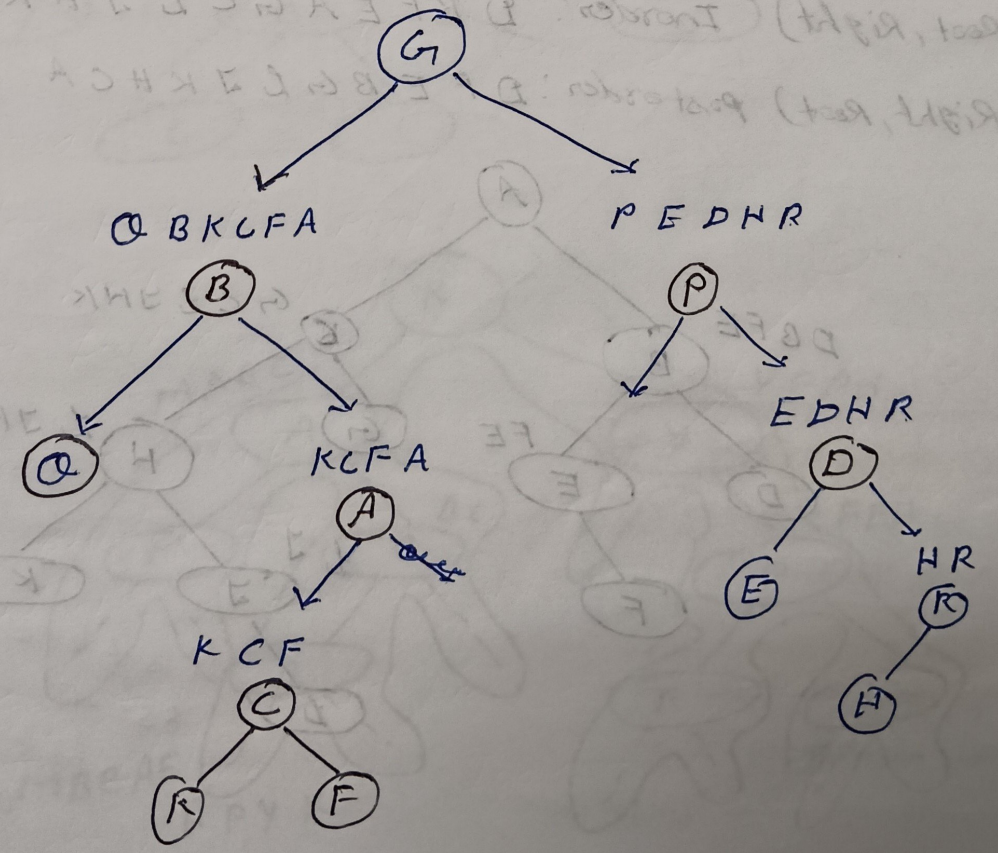
Pre-order + Post-order ⇒ unique binary tree नहीं बनेगा

→ Binary Tree construction from given preorder & Inorder

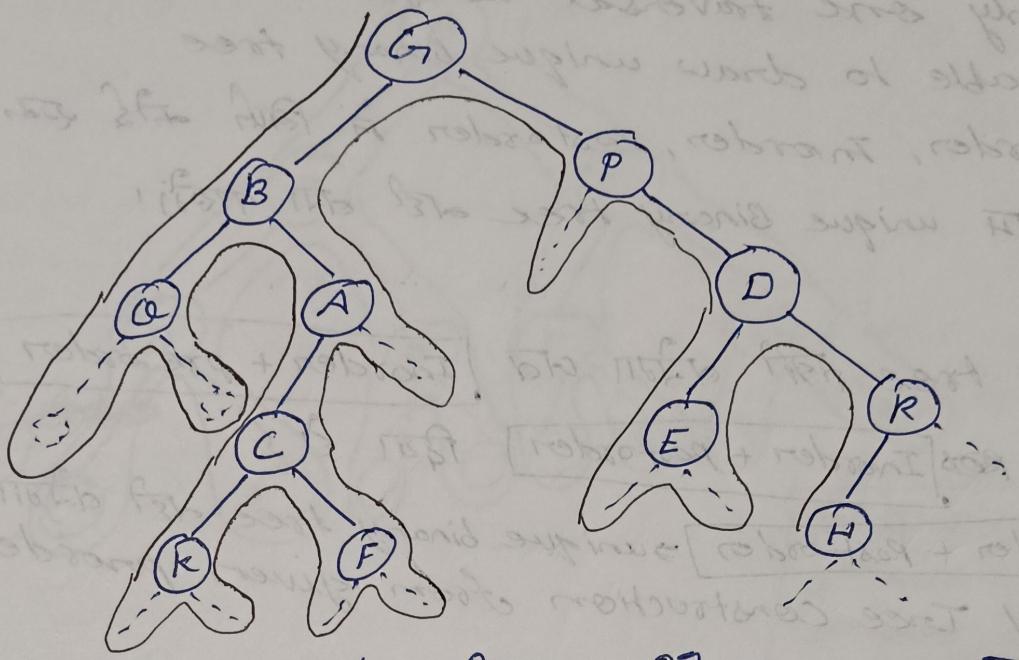
(Left, Root, Right) Inorder: Q, B, K, C, F, A, G, P, E, D, H, R

(Root, Left Right) Pre-order: G, B, Q, A, C, K, F, P, D, E, R, H

Preorder से root मिल जाएगा, Inorder से left और Right का node मिल जाएगा।
pre-order में traverse करेंगे जो node पहले आएगा वो root बन जाएगा।



so, tree will look like



इस Tree का pre-order निकाल लेते है same आना चाहिए

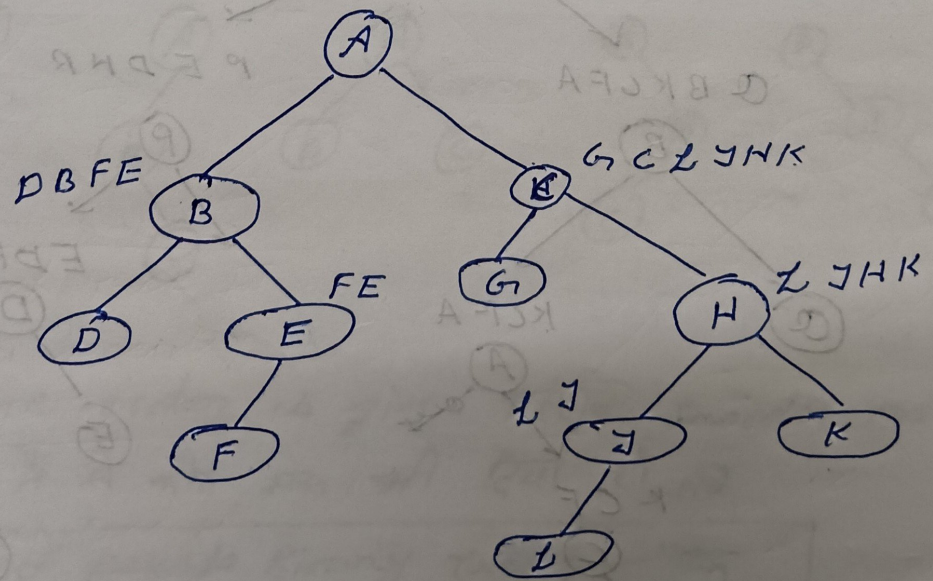
Pre-order: GBQACKFAPDERH ⇒ same आ रहा है

→ Construct Binary Tree from Postorder & Inorder Traversal

Postorder में Root ~~left~~ Right में रहता है तो यहाँ Right से देखेंगे। जो node पहले आएगा root node.

(Left, Root, Right) Inorder: D B F E A G C L J H K

(Left, Right, Root) Postorder: D F E B G L J K H C A



Q

1 M B C A F H P Y K

2 K A M C B Y P F H

3 M A B C K Y F P H

Find preorder, Post-order, Inorder?

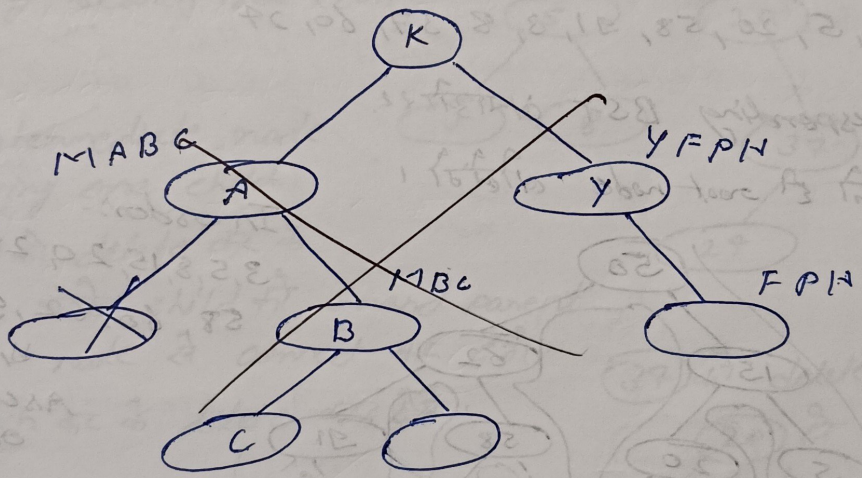
** Preorder में Root node पहले होता है & post order में Root node last में इसी का use करेंगे

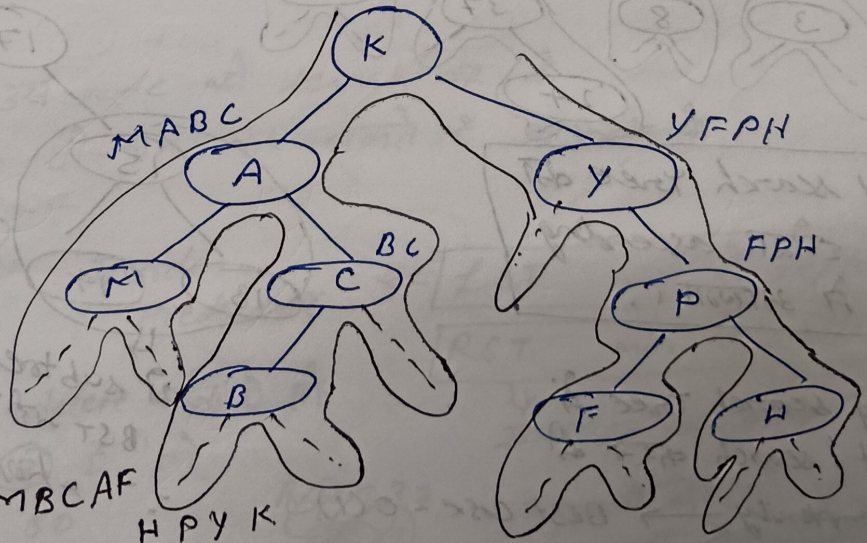
2 वाला pre-order है

1 वाला Post-order है।

Preorder:- K A M C B Y P F H

In-order : M A B C K Y F P H





Postorder: MBCAF HPYK

Post order: M B C A F H P Y K

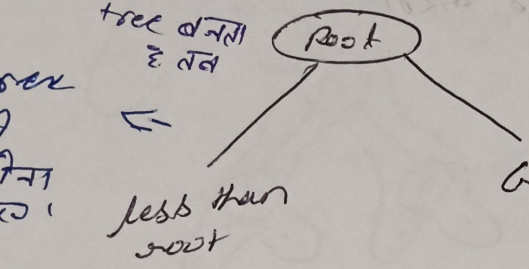
Binary Search Tree

n-element से BST बनाने का Time complexity
Best case - $O(n \log n)$
worst case - $O(n^2)$

↓ जब skew tree बनता है तब

Time complexity of searching in Binary Tree $O(n)$

↓ इसी को कम करने के लिए Binary search tree को introduce किया।



B.S.T

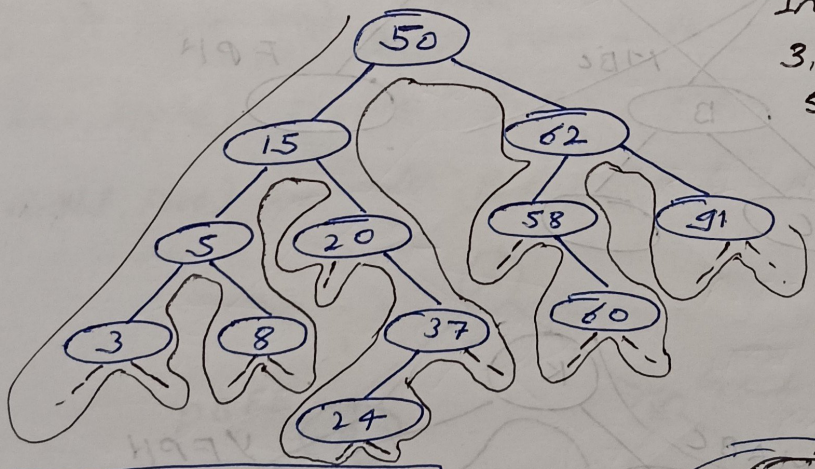
Sub-tree में भी ऐसा होना चाहिए।

अगर मान लो equal है तब left या right में place कर दो but अगर left में कर रहे हो तो throughout whole construction left में रखना है

50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24

↓ इसके corresponding BST बनाओ।।

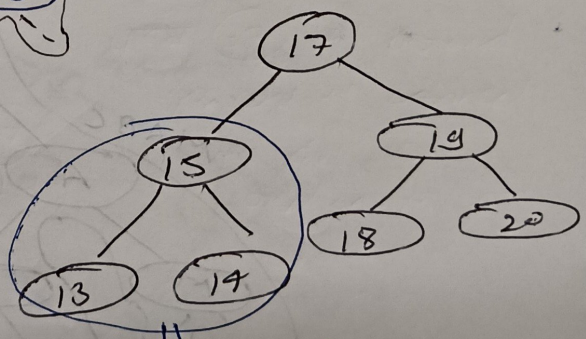
पहले node को ही root node बोलेंगे।



In order:-
3, 5, 8, 15, 20, 24, 37, 50, 58, 60, 62, 91

↓ Ascending order में है

** Binary search tree की In-order हमेशा ascending order में आएगा।

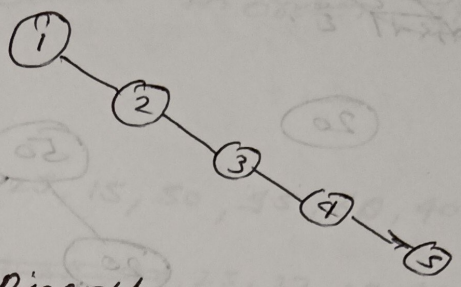


↓ ये subtree BST नहीं है ∴ whole tree is not BST

Binary search tree में Element search करने की Time Complexity →
- Best case - $O(1)$
- Average case $O(\log n)$
- Worst case $O(n)$

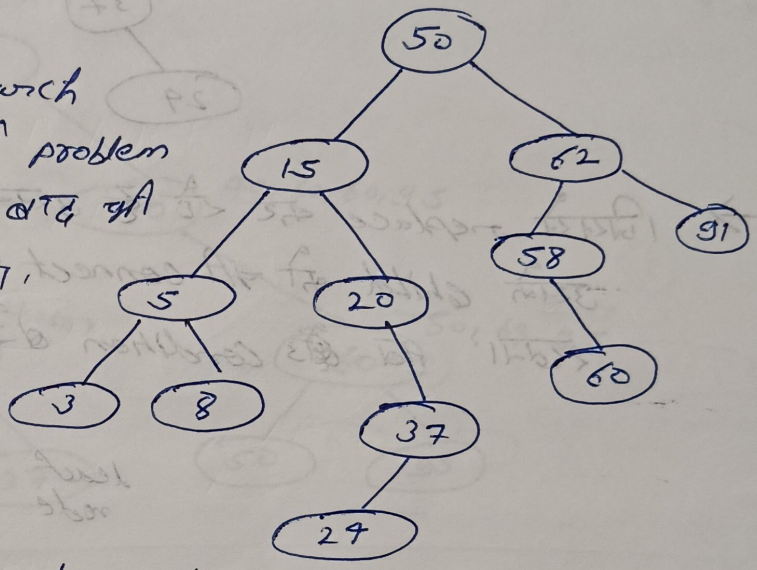
worst case - O(n) why? Element ascending or descending order में अगर दिए हो तो Right skewed tree बनता है या Left skewed tree बनता है इसलिए।

1, 2, 3, 4, 5 → B.S.T ⇒



→ Deletion operation in Binary search tree

1) Leaf node - simple search कर के delete कर देंगे कोई problem नहीं होगा delete करने के बाद भी Binary search tree रहेगा।



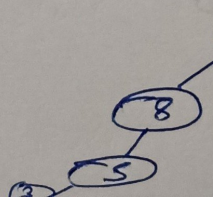
2) Intermediate node having one child.
उसको delete कर के उसके child को Grand parent node से connect कर देंगे।
Search कर के delete कर देते।

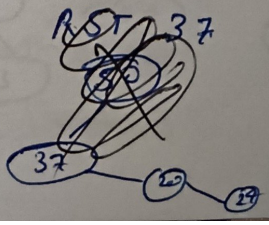
(37) को delete करना है तो (20) से (24) को connect कर दो
Grand parent
child

3) Intermediate node having two child
उस node को search कर के दो काम कर सकते हैं उस node के

Inorder predecessor ⇐ LST max से replace कर दें
RST min से replace कर दें ⇒ Inorder successor

15 को delete करना है तो
LST = (8) है तो

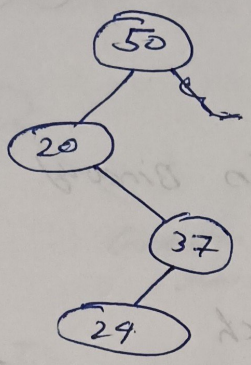




37 को delete करना है तो भी जोड़ करना होगा। तो उसे उसके Grandparent से जोड़ देंगे

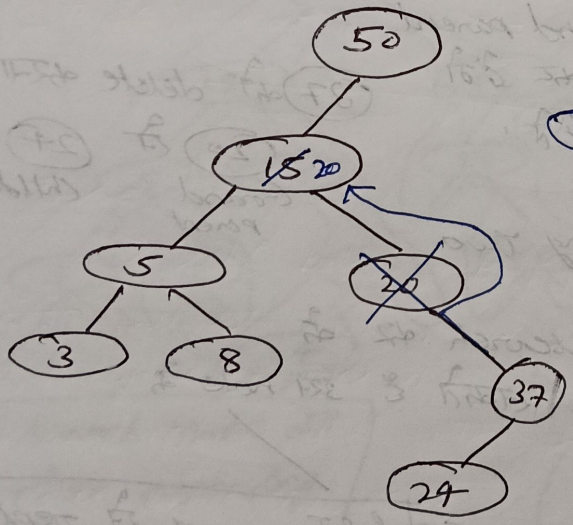
simple है जब जिसको replace कर रहे हैं वो भी node delete हो रहा है तो उसको भी connect करना है।

RST (20)



→ जिसको replace कर रहे हैं उसको जब हटाएंगे तो उसके child को भी connect करना होगा इसका ध्यान रखना। तब 3 condition बनेगी जिसको replace कर रहे हैं वो

leaf node | with 2 child होगा | with 1 child होगा।



(15) को delete करना है तो

RST = 20 ⇒ min

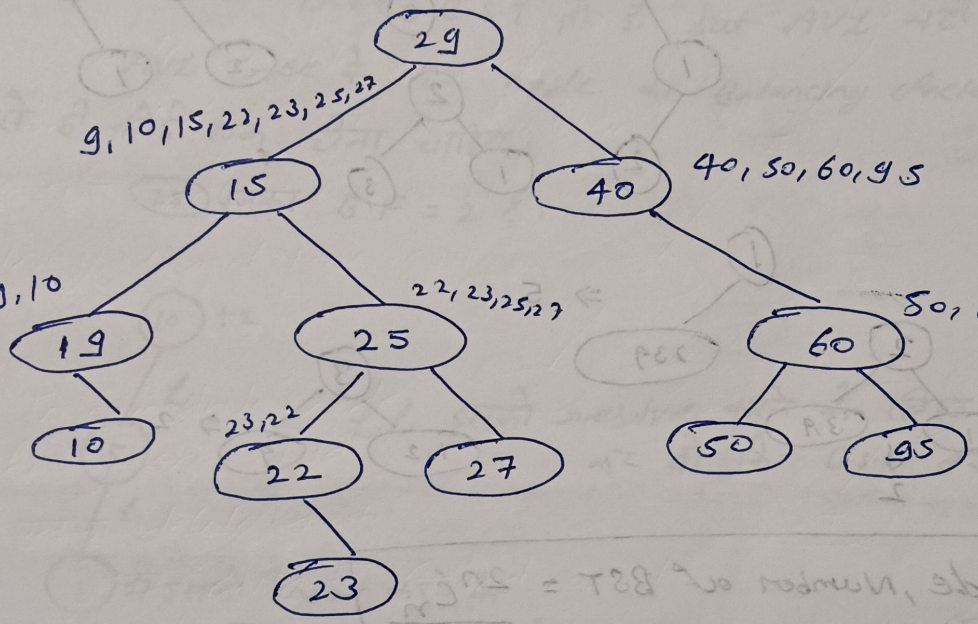
पहले (37) को Inorder parent से connect कर देंगे फिर replace कर देंगे 20 से

→ Construct Binary search tree

Preorder, या Post-order दिया है तो unique Binary search tree बना लेंगे because Inorder में Ascending order में होता है।

Q: Postorder: 10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29

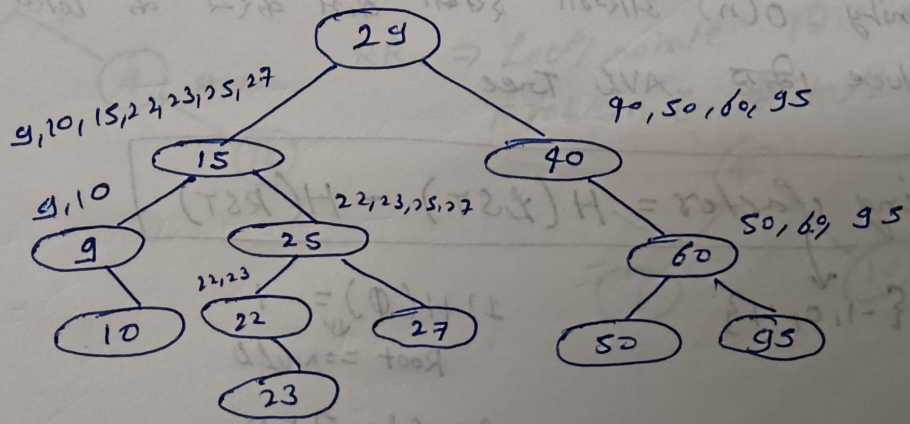
Inorder: 9, 10, 15, 22, 23, ~~27, 29, 40~~, 25, 27, 29, 40, 50, 60, 95



* letter दिया है तो (G, A, B, C, D) इस तरह तो उसका inorder letters को alphabetical order में लिख देंगे।

Q Preorder: 29, 15, 9, 10, 25, 22, 23, 27, 40, 60, 50, 95

Inorder: 9, 10, ~~22~~ 15, 22, 23, 25, 27, 29, 40, 50, 60, 95



→ How many different BST can be constructed using 'n' distinct keys?

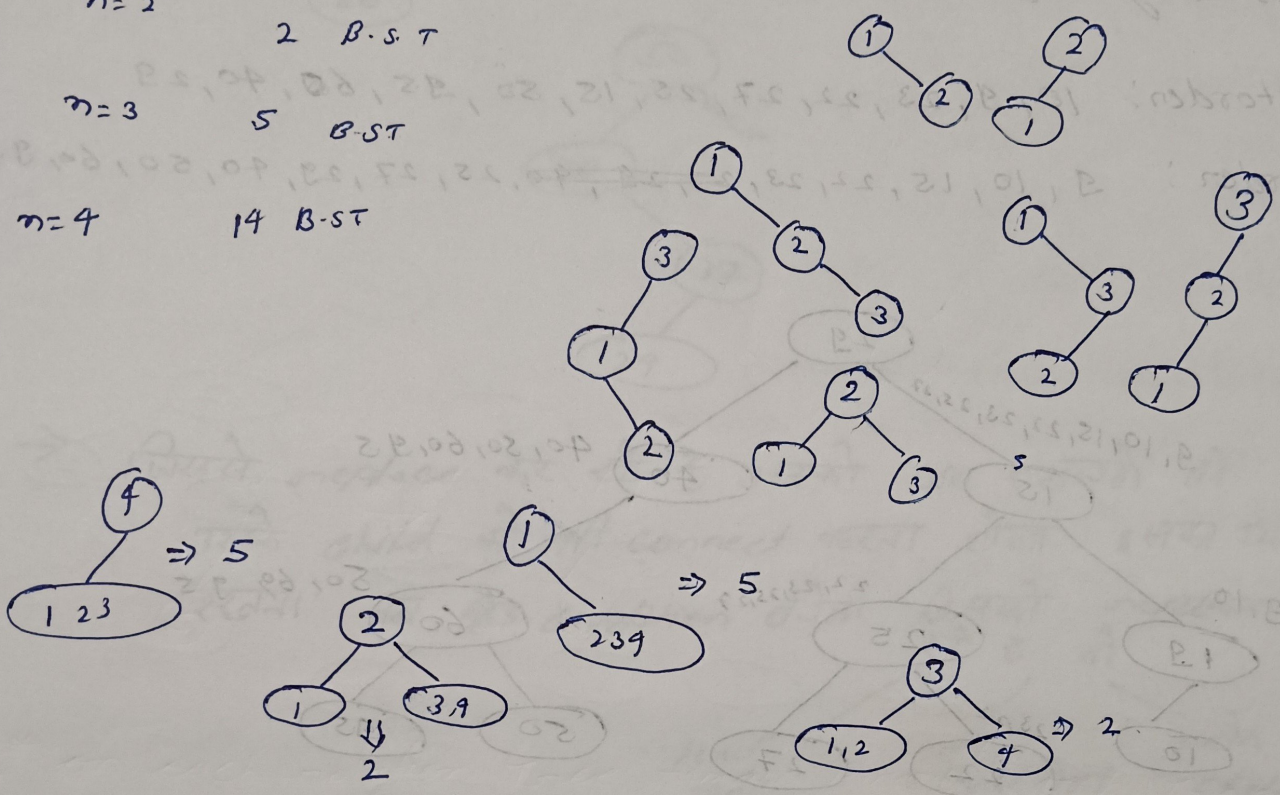
$n = 0$ → 1 ⇒ Empty Binary search tree

$n = 1$ → 1 B.S.T

$n = 2$ → 2 B.S.T

$n = 3$ → 5 B.S.T

$n = 4$ → 14 B.S.T



From n node, Number of BST = $\frac{2nC_n}{(n+1)}$

→ **AVL Tree:**

⇒ Self Balancing BST

इसकी जरूरत क्यों पड़ी 1,2,3,4,5 ⇒ इसके corresponding skew Binary search tree बनेगा उसे search करने की Time Complexity O(n) आएगा इसको कम करने के लिए हम लोग introduce किए AVL Tree

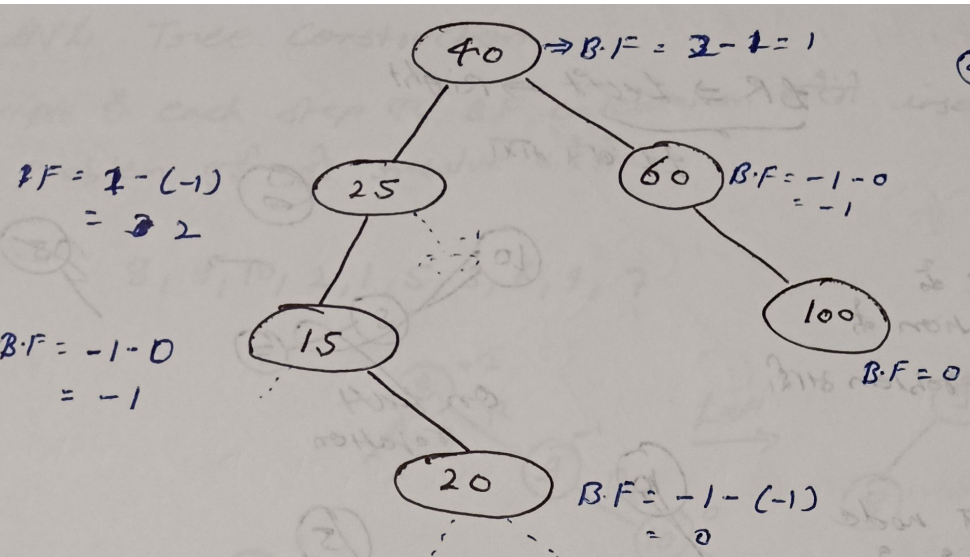
Balancing factor = H(LST) − H(RST)

↓

{-1, 0, +1}

1) $H(\phi) = -1$ ↓ Root == NULL

2) H(leaf) = 0

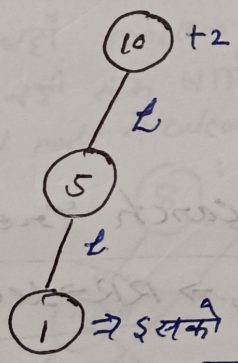


(40) की Height निकालना है तो (25) की निकाल लो और (B)

ये Tree B.S.T भी है but AVL नहीं है because AVL Tree के हर node का Balancing factor {-1, 0, 1} इसमें से ही कोई एक होना चाहिए।

(25) इसका B.F = 2 है।

→

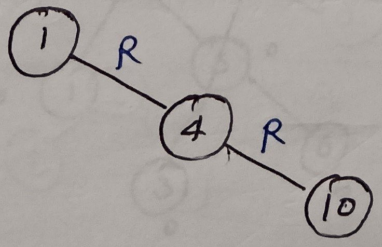


⇒ इसको insert करने के बाद problem हुआ है अब यहाँ से Root तक जायेंगे फिर वहाँ जहाँ problem हुई है फिर वहाँ ~~root~~ से इस node तक आयेंगे First ~~last~~ 2 node को देखेंगे यहाँ First ~~last~~ 2 node Left में है तो L-L

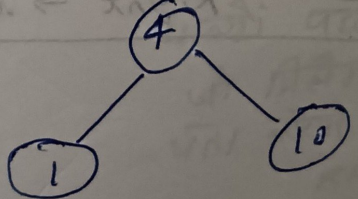
LL इसको resolve करने के लिए Right rotate कर दो कि B.S.T बना रहेगा।

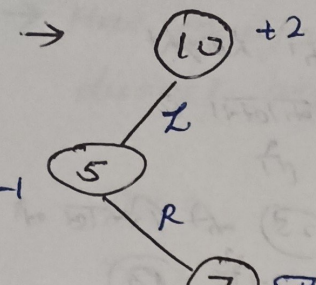
जिस node में problem है उसे left rotate कर देंगे

→



RR ⇒ Left rotate जिस node में problem है उसे left में rotate कर दो

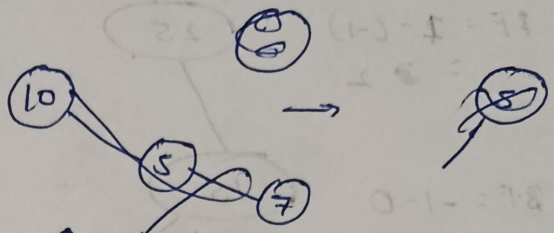




LR ⇒ Left → Right

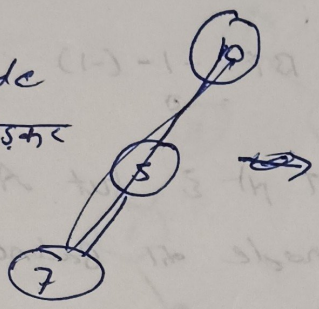
LL कर गया

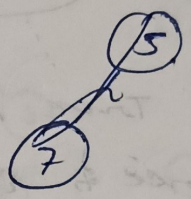
इस node के insertion के बाद problem आई।

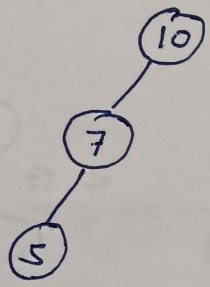


On Left rotation

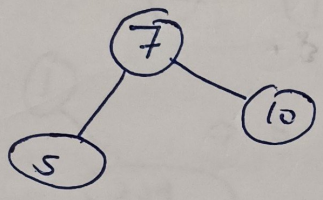
Left rotation जिस node में problem है उसे छोड़कर करेंगे.







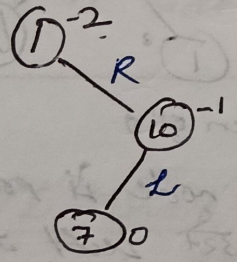
Right →



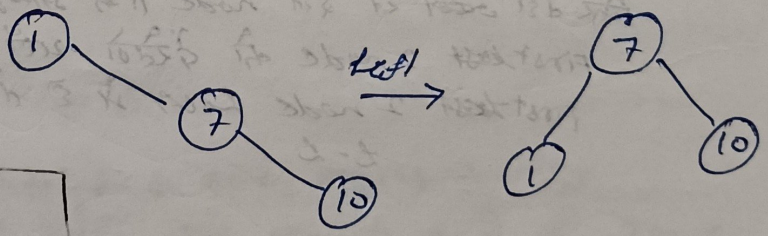
** on each rotation Tree remains Binary search Tree

RL → RR → Left

→



RL → Right → Left

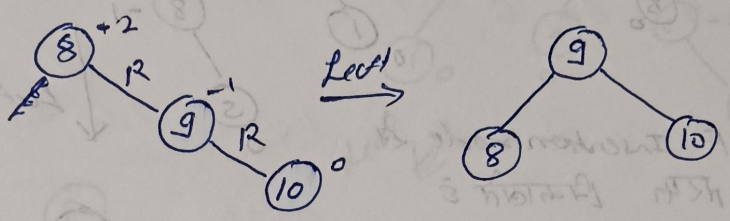


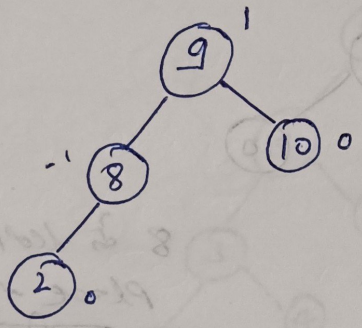
LL = ~~Left~~ Right
RR = B left
LR = Left → Right
RL = Right → left

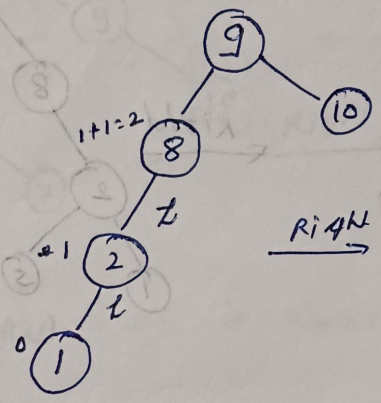
→ AVL Tree Construction

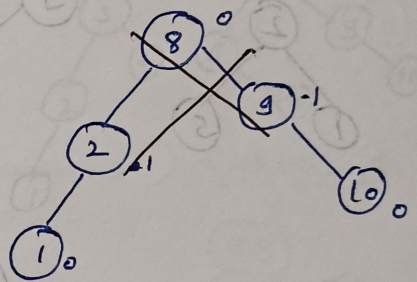
Simple है each step पर B.F. calculation करेंगे insertion के बाद problem हुई तो resolve करेंगे।

8, 9, 10, 2, 1, 5, 3, 6, 4, 7

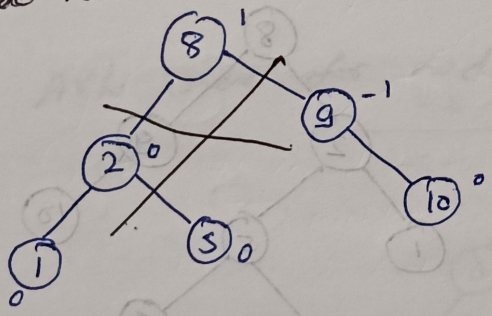


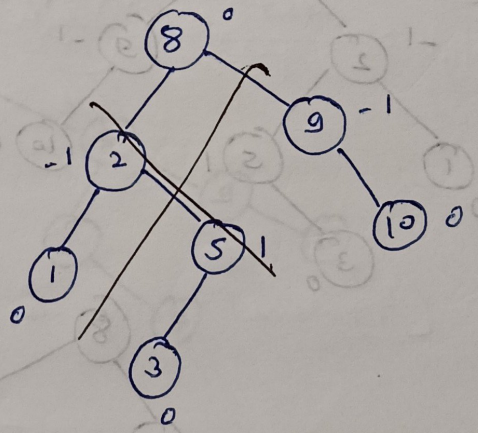


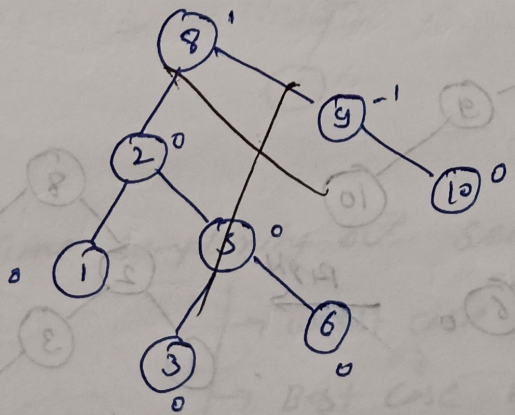




Note:- जहाँ problem create हुआ वहीं से नीचे जाओ leaf node से calculate करेंगे।

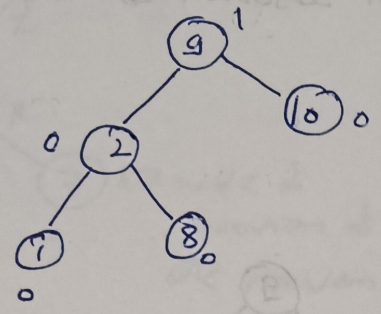


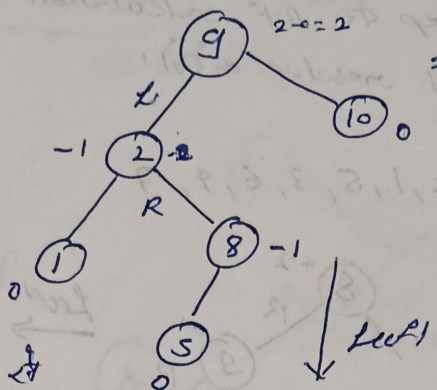




→ जहाँ problem हुई वहाँ से ~~created node~~ newly inserted node की नजर जाएगी पहले 2 node जो मिलेगा उसी से पता करेंगे LL, LR, RL, RR है

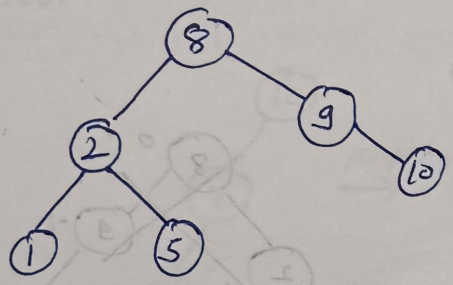
8, 9, 10, 2, 1, 5, 3, 6, 9, 7

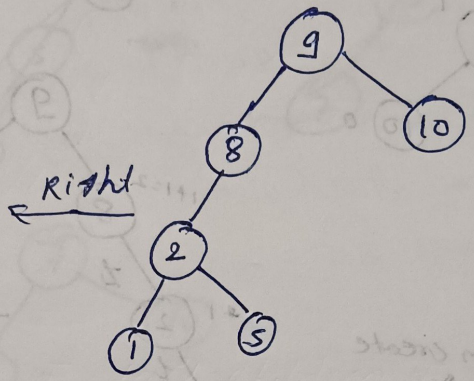




⇒ यहाँ पर LR problem है first two movement देखना

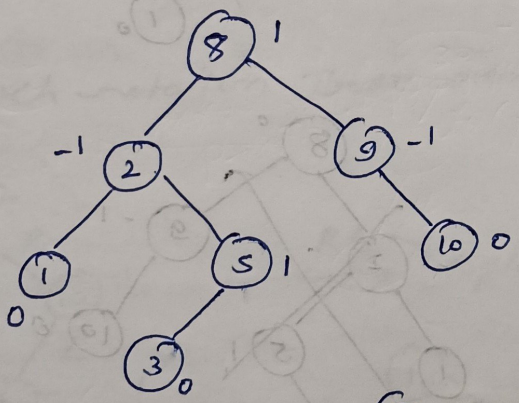
* हमेशा B.F Insertion node से root की तरफ निकालते है

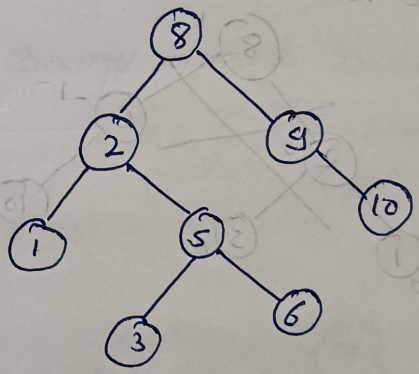


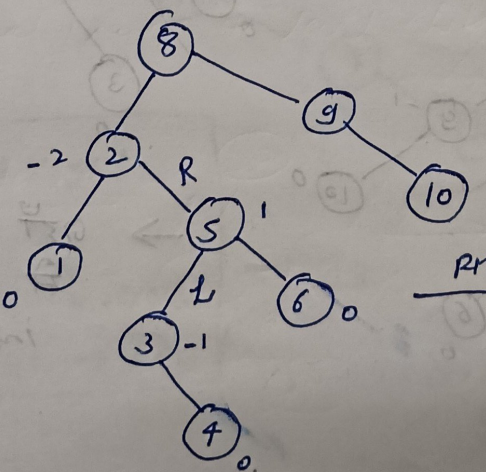


Right

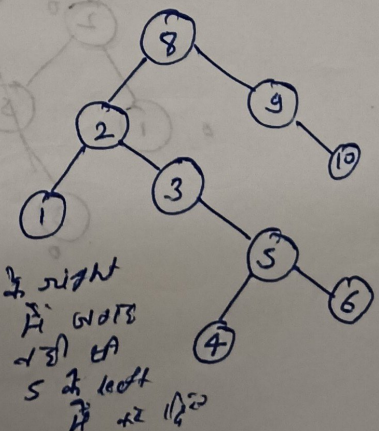
8 के left में place नहीं था इसलिए 2 के right में रख दिए



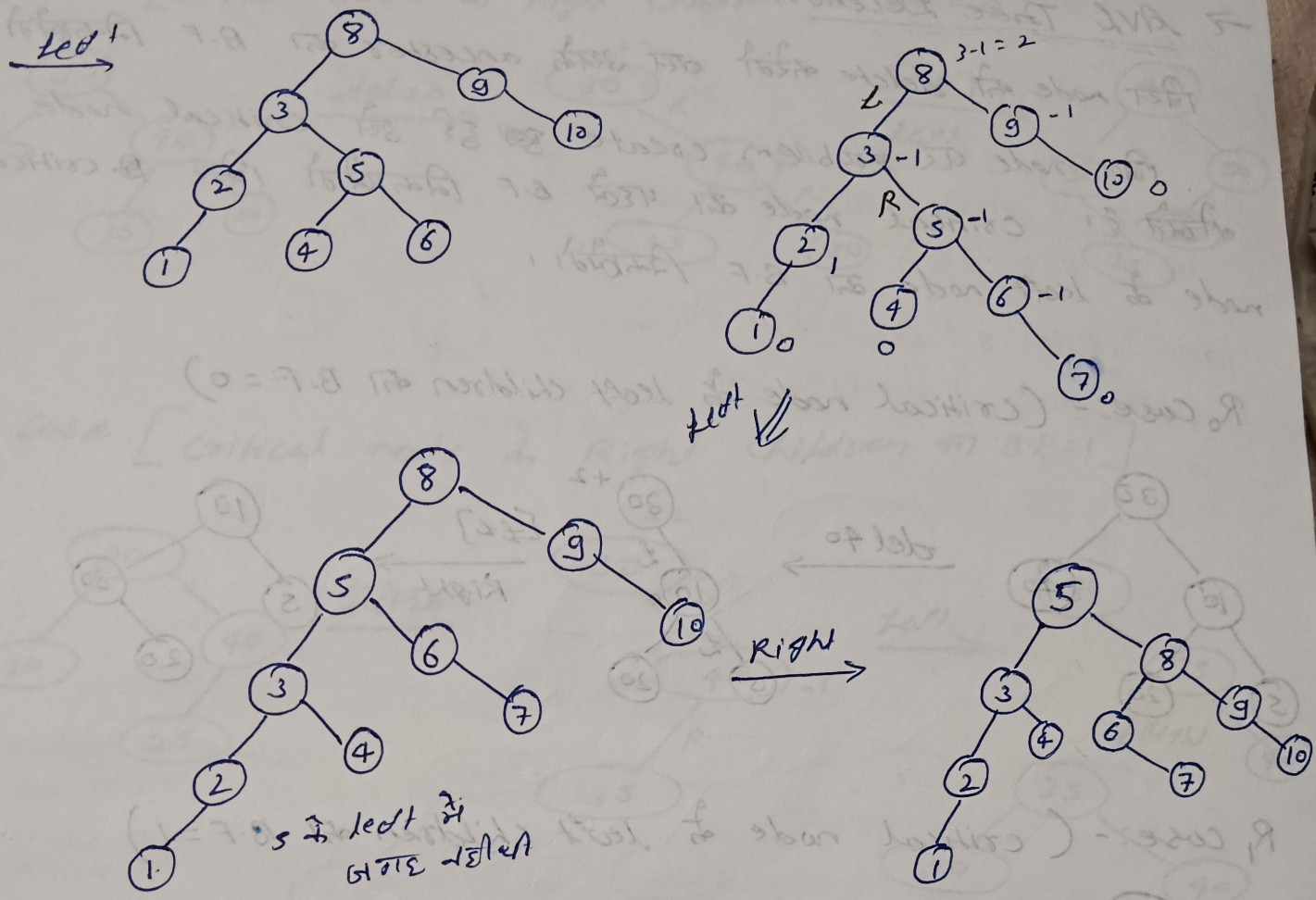




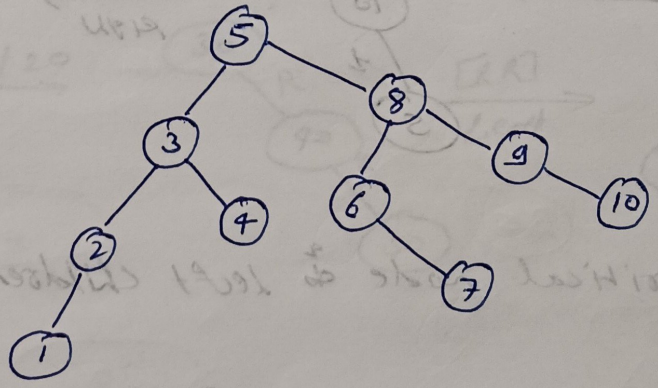
Right



3 के right में node नहीं था 5 के left में रख दिए



b) AVL Tree for node: 8, 9, 10, 2, 1, 5, 3, 6, 4, 7



Time complexity of search in AVL Tree

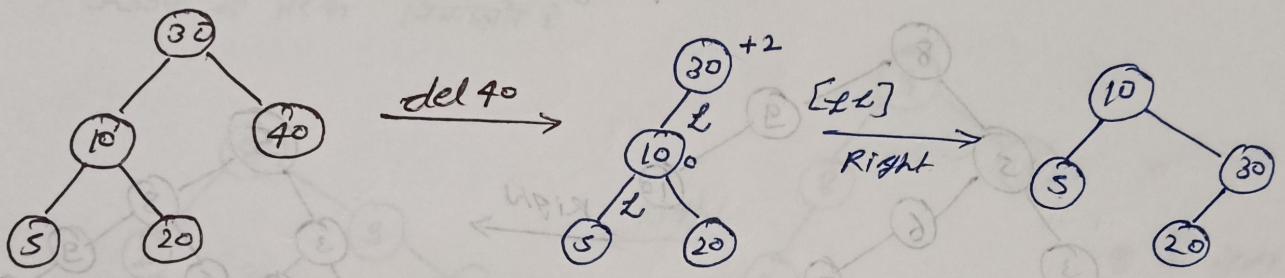
- worst case O(log n)
- Best case O(1)
- Average case O(log n)

→ AVL Tree Deletion

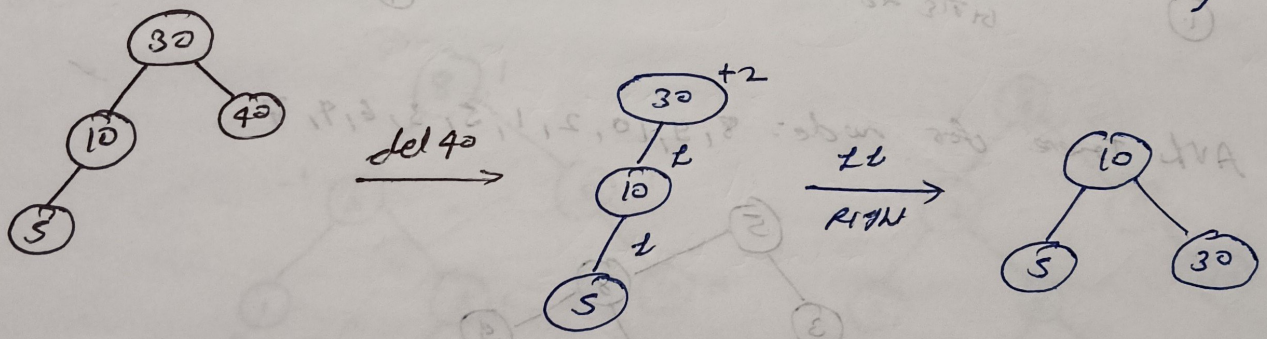
जिस node को delete करेंगे बस उसके ancestor का B.F निकालेंगे.

जिस node पर problem create हुई उसे critical node बोलते है। critical node का पहले B.F निकालेंगे फिर critical node के left node का B.F निकालेंगे।

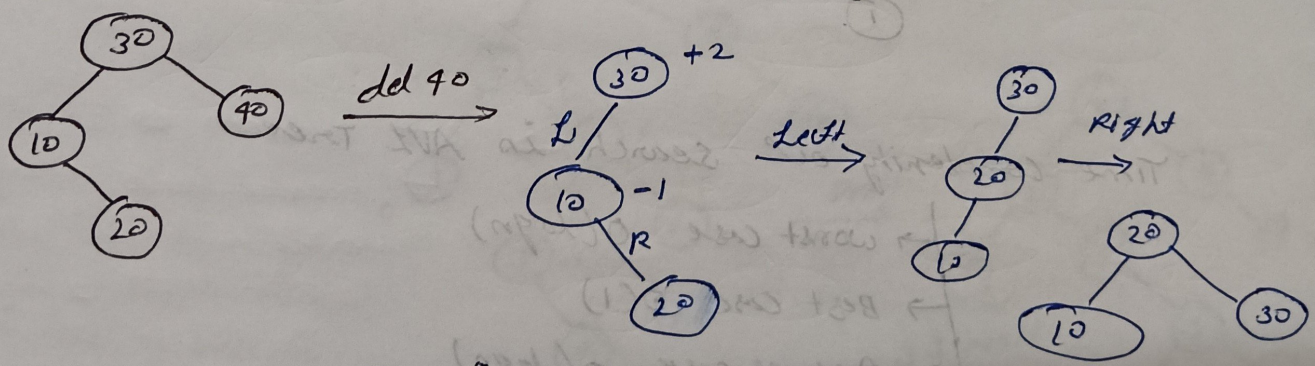
R₀ case:- (critical node के left children का B.F = 0)



R₁ case:- (critical node के left children का B.F = 1)

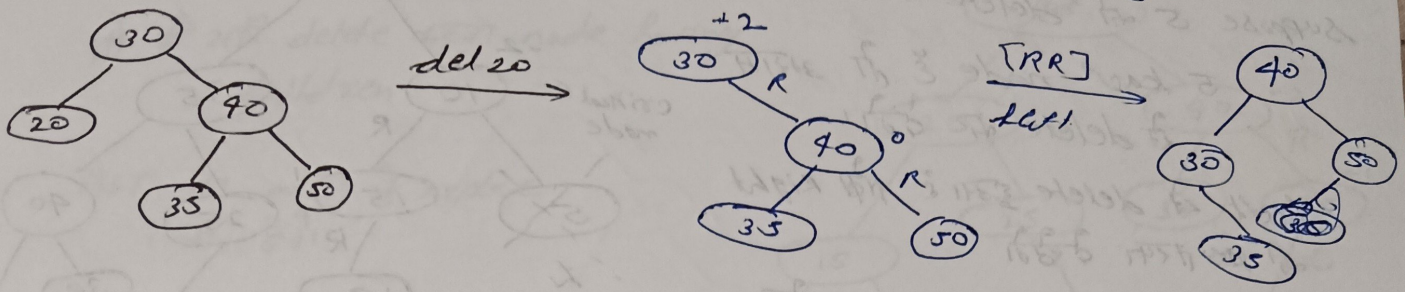


R-1 case:- (critical node के left children का B.F = -1)

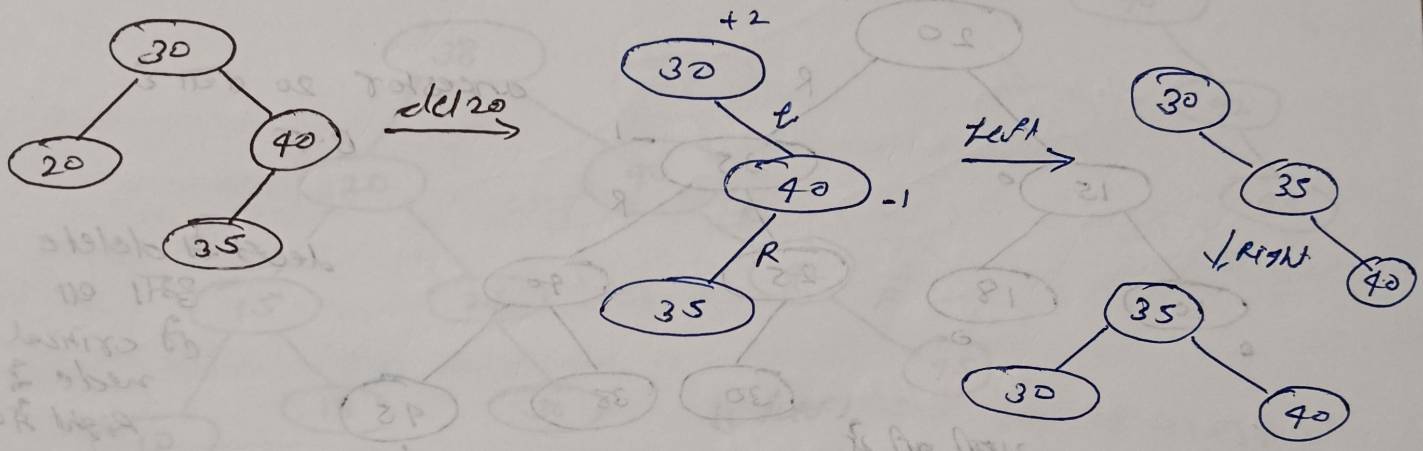


** Right से delete किया तो इसलिए left का देखते अगर left से delete करेंगे तो Right का देखेंगे।

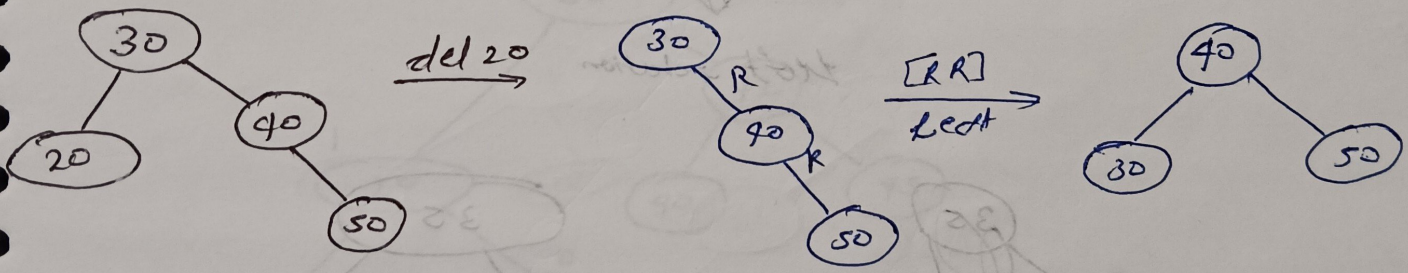
L_0 case [critical node के Right children का B.F = 0]



L_1 case [critical node के Right children का B.F = 1]



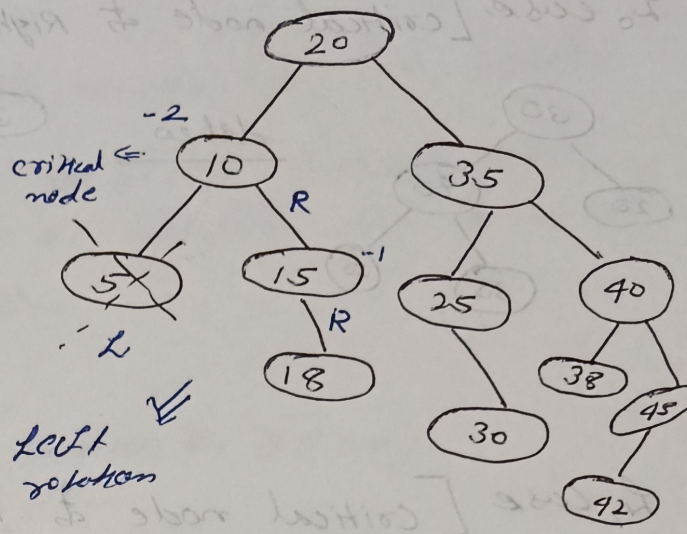
L_{-1} case [critical node के Right children का B.F = -1]



Suppose 5 को delete करना है

5 leaf node है तो आराम से delete कर देंगे।

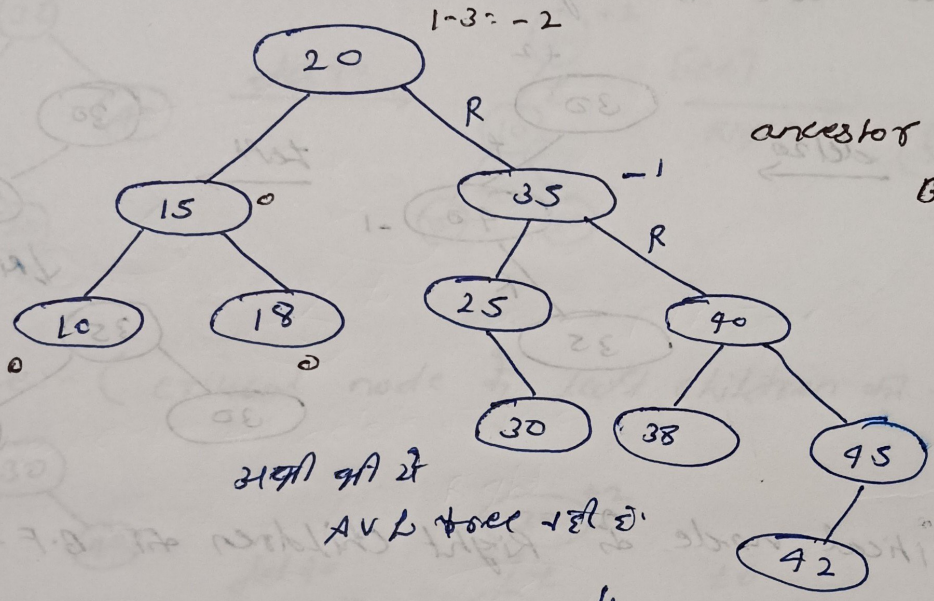
left से delete हुआ है तो right की तरफ देखेंगे



left rotation

कितना कितना B.F check करना होगा

critical node के ancestor का root node तक



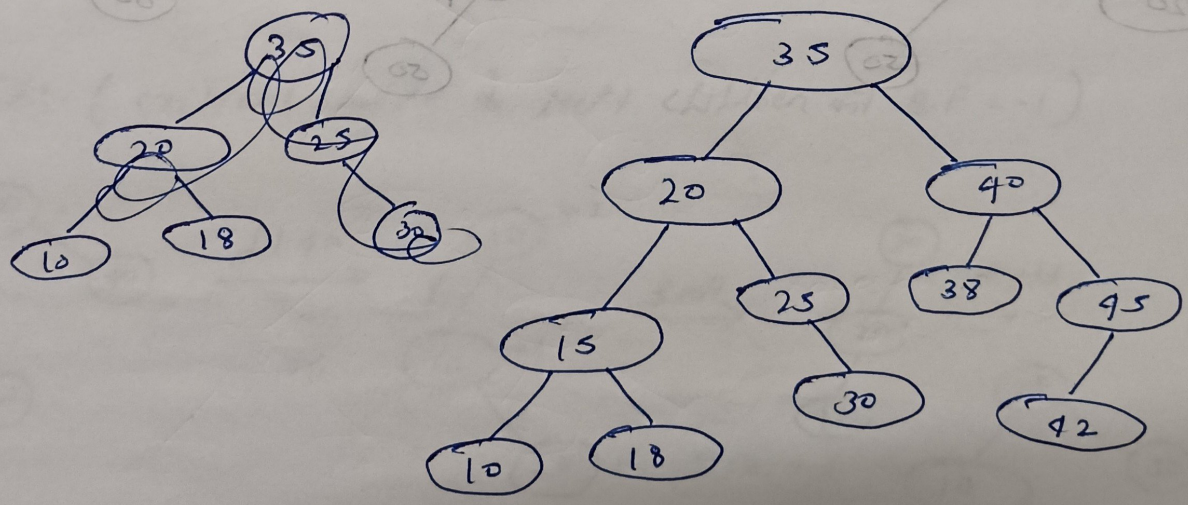
ancestor 20 बचा है

B.F

left से delete हुआ था तो critical node के right में आया

अभी भी ये AVL tree नहीं है

left rotation

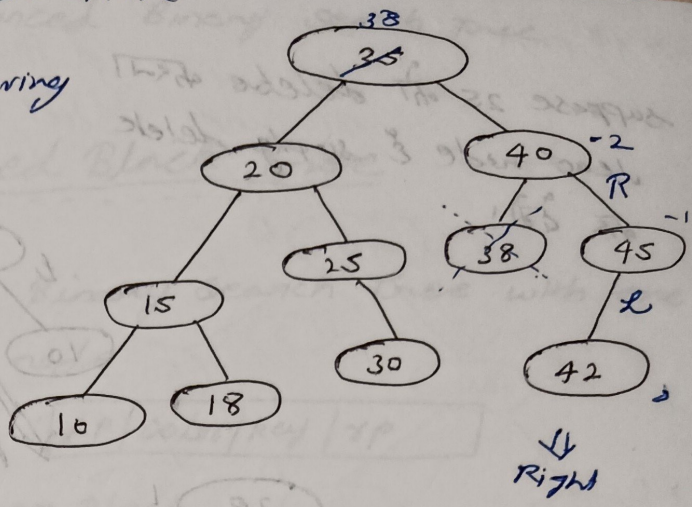


सारे ancestor check कर लिए अब ये AVL tree है।

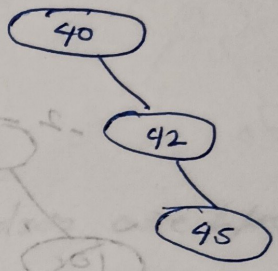
suppose root node को delete करना है।

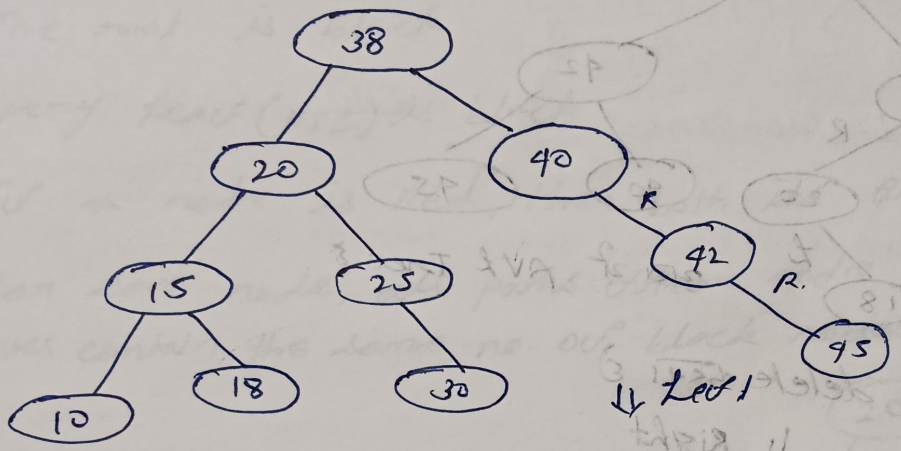
तो फिर यहाँ delete करना node having 2 children

RST का min से replace कर देंगे।

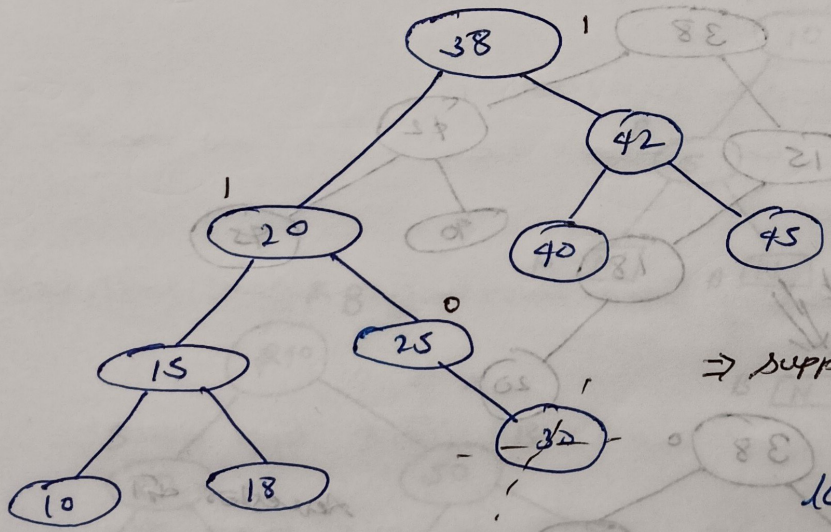


⇓ Right





⇓ Left



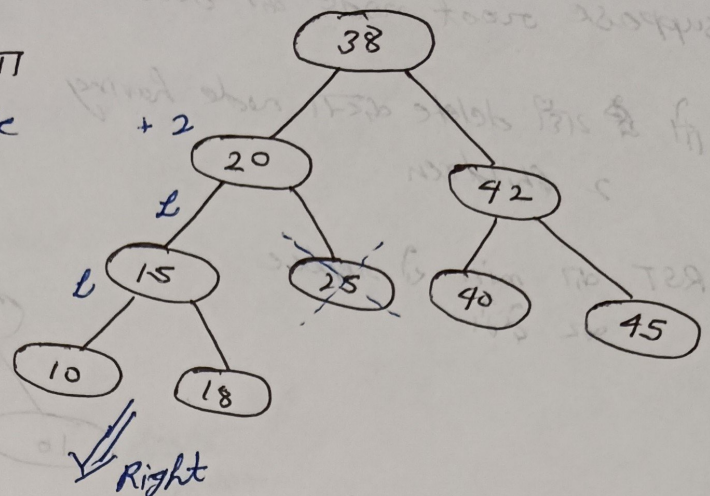
⇒ suppose इसमें 30 को delete करना है तो

leaf node है simply delete कर देंगे

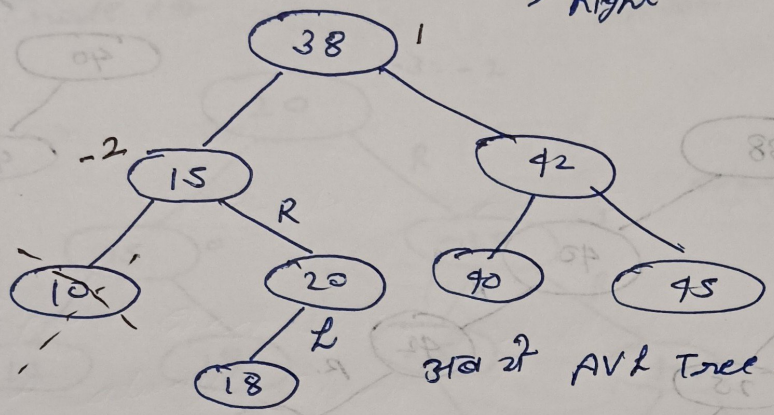
अब Ancestor का check करेंगे B.F

delete करने के बाद भी AVL Tree है।

suppose 25 को delete करना
lead node है simply delete
कर देंगे।



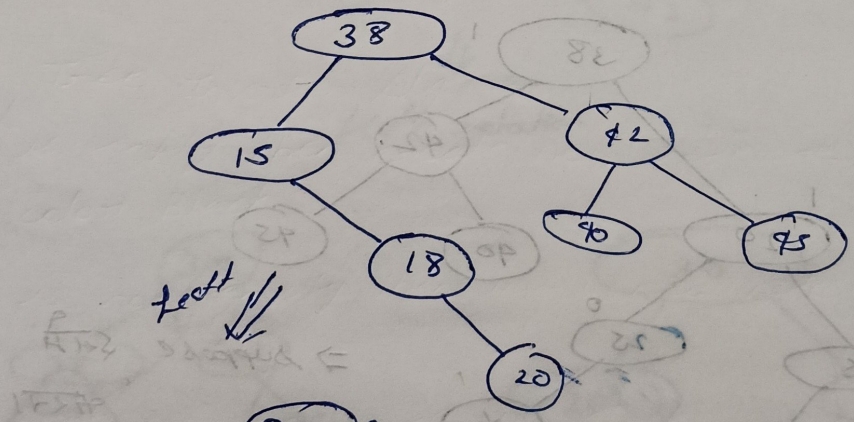
Right



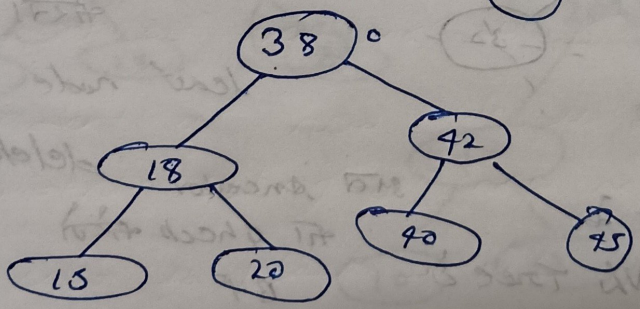
अब ये AVL Tree है

suppose 10 को delete करना है

Right



left



Ancestor की
B.F = 0

B.ST ⇒ searching, delction में worst case Time complexity
O(n)

AVL Tree में worst case Time complexity में searching,
deletion के लिए O(logn)

* AVL Tree, ~~Binary~~ Balanced Binary Search Tree

Introduction to Red Black Tree

A Red Black Tree is a Binary Search Tree with one extra bit of storage per node

lp	color	key	rp

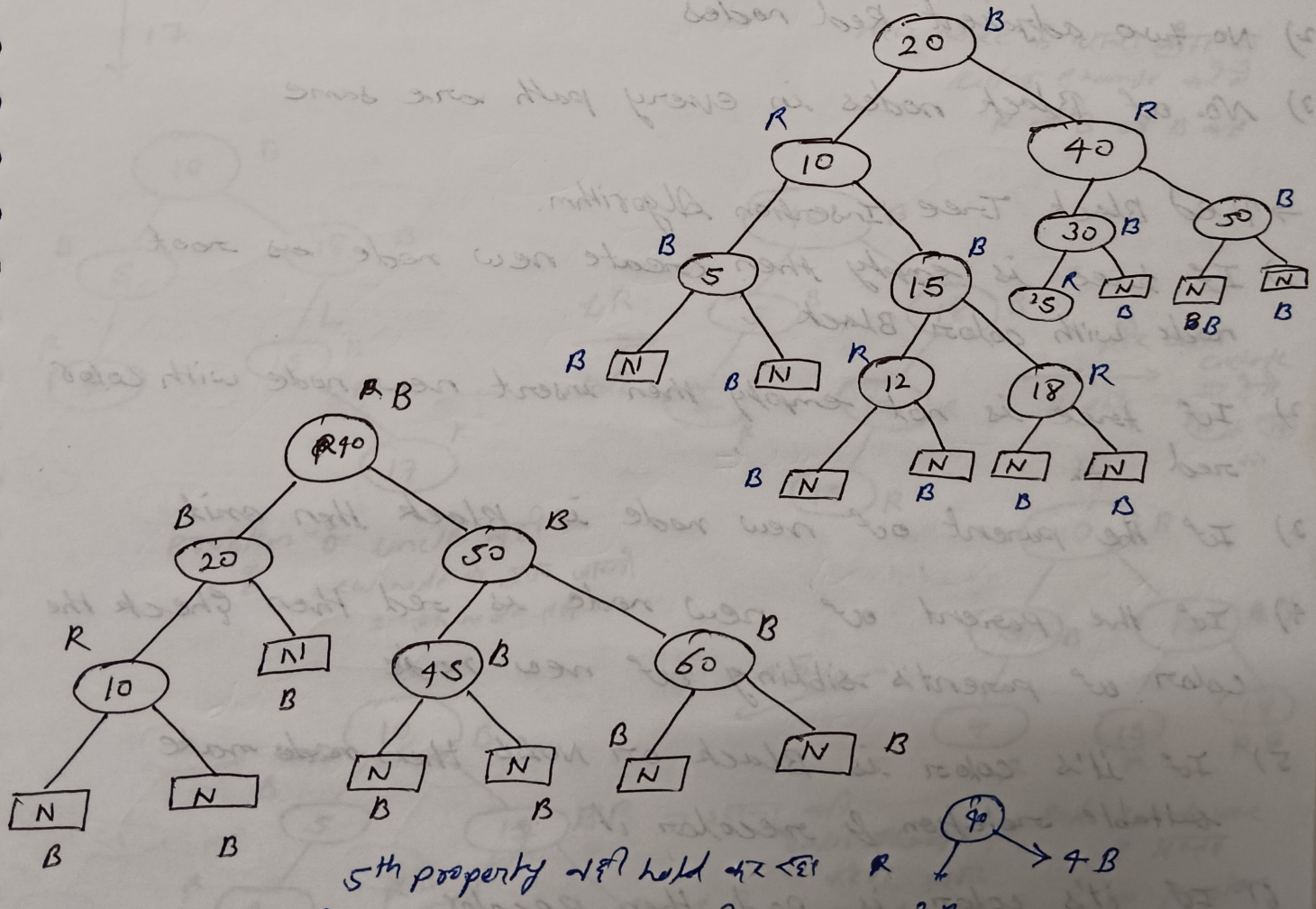
1.) Every node is either Red or Black

2.) The root is black

3.) Every leaf (NIL) is black

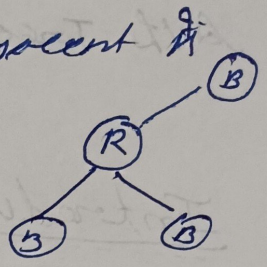
4) If a node is Red, then both its children are black

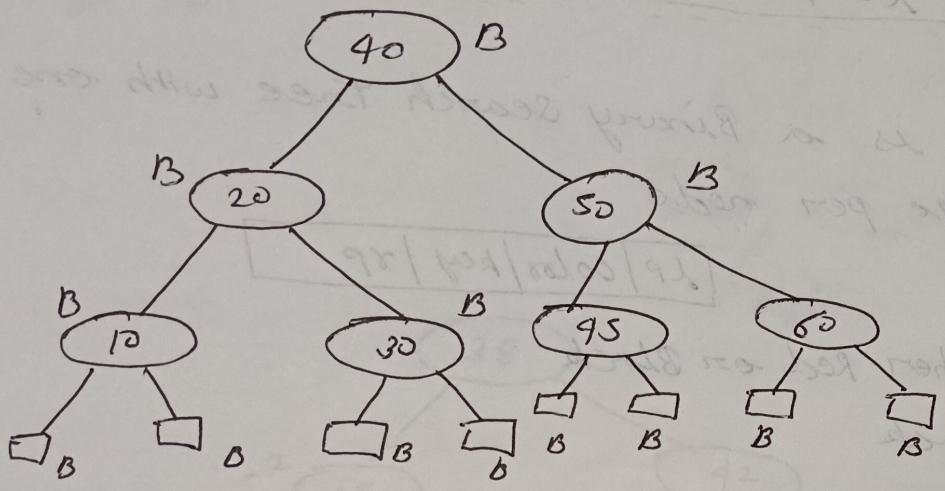
5.) For each node, all paths from node to descendant leaves contain the same no. of black nodes



→ 4th property का conclusion है

कोई भी node R मिला तो उसके adjacent में सारे node Black होने चाहिए





Red-Black Tree 5 property satisfy कर रहा है।

Mainly 3 point check करना है Red-Black tree के लिए।

1) Root is always Black.

2) No two adjacent Red nodes

3) No. of Black nodes in every path are same

→ Red Black Tree Insertion Algorithm.

1) If tree is empty then create new node as root node with color Black

2) If tree is not empty then insert new node with color red

3) If the parent of new node is Black then exit

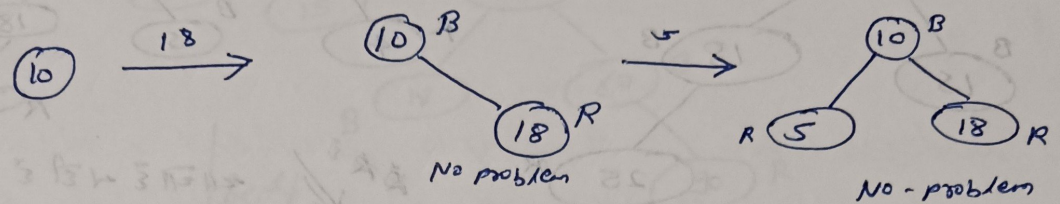
4) If the parent of new node is red then check the color of parent's sibling of new node

5) If it's color is Black or NULL then ~~node~~ make suitable rotation & recolor it.

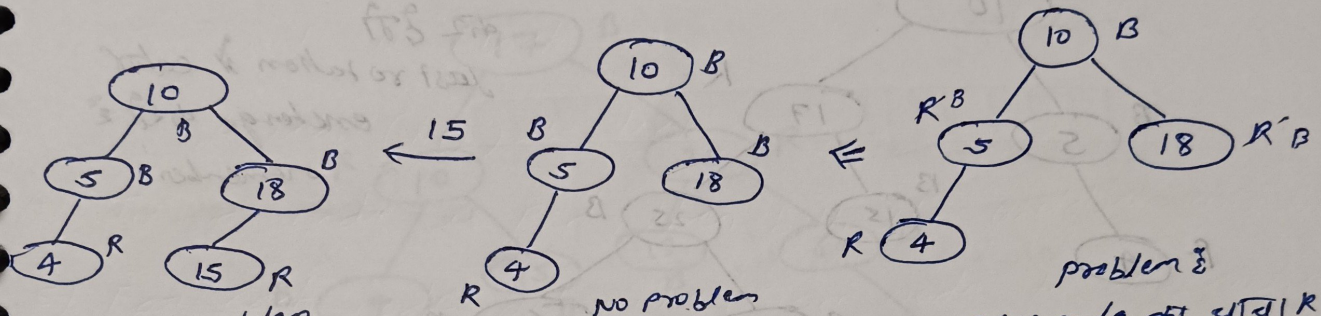
6) If it's color is Red then Recolor.

10, 18, 5, 4, 15, 17, 25, 60, 1, 90

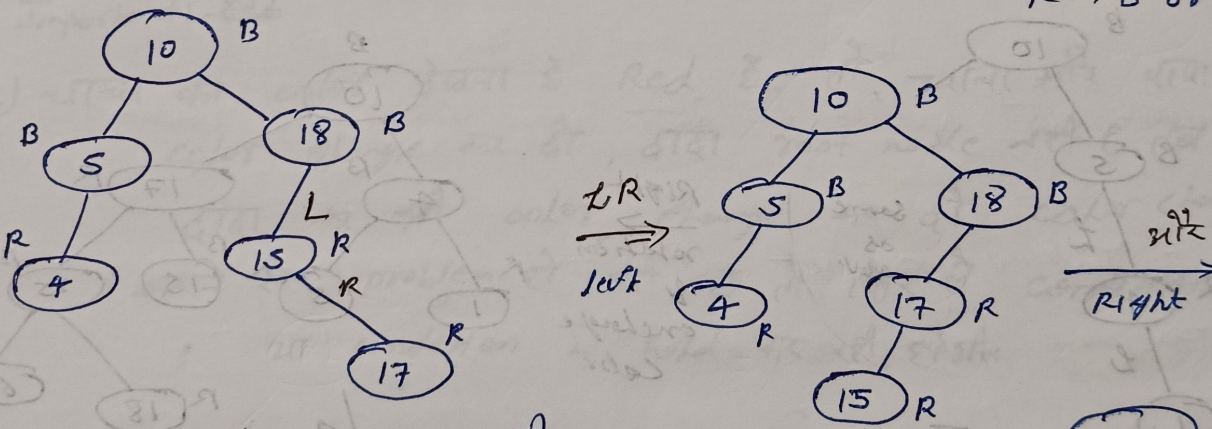
Red Black tree एक तरह का BST है

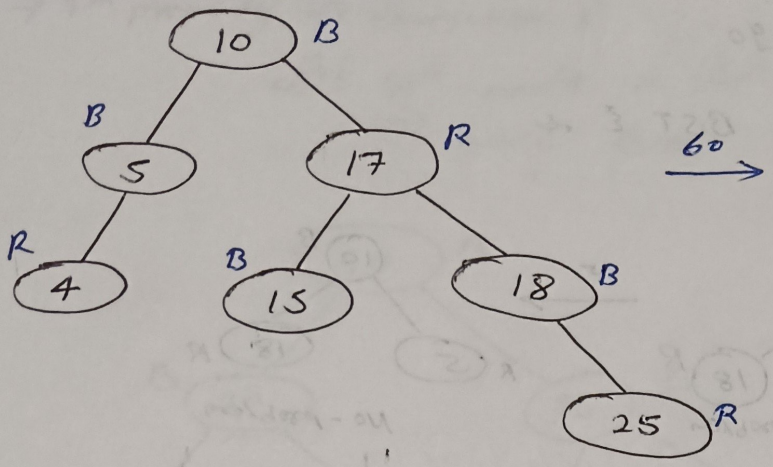


4

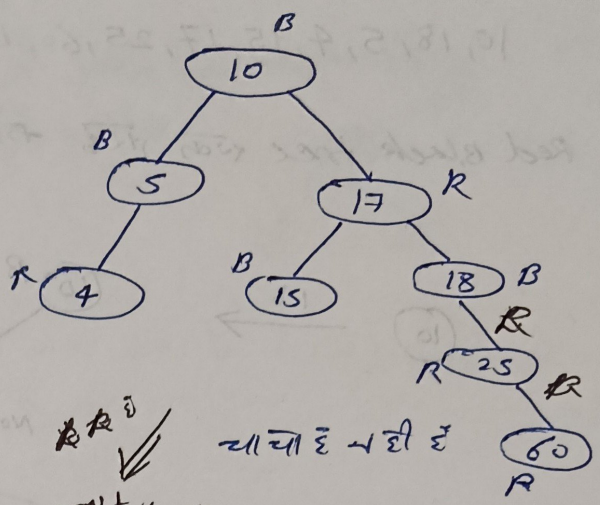


17





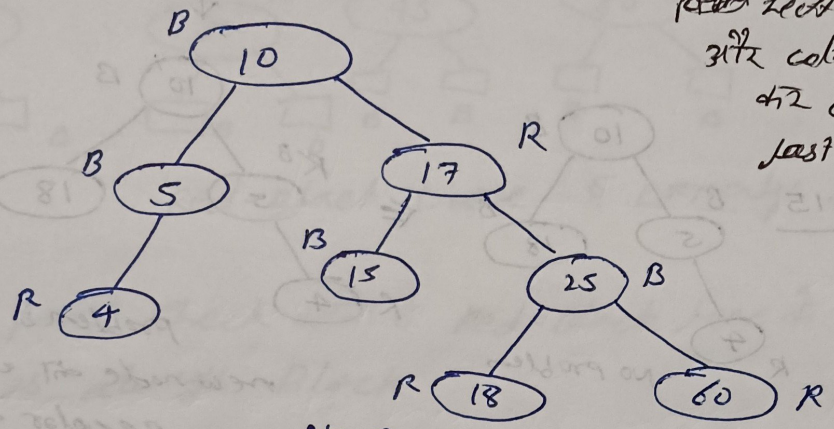
60 →



चाचा है नहीं है

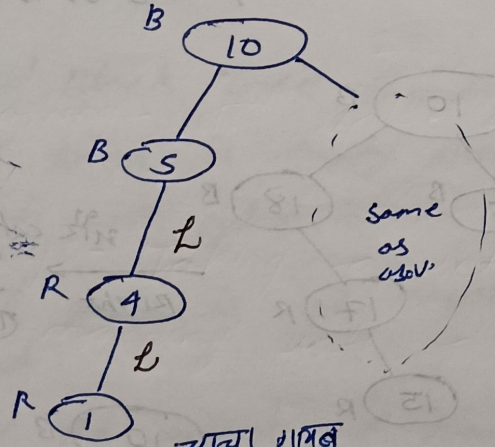
Right rotation और color exchange कर देंगे

least rotation में color exchange करते हैं remember



No Problem

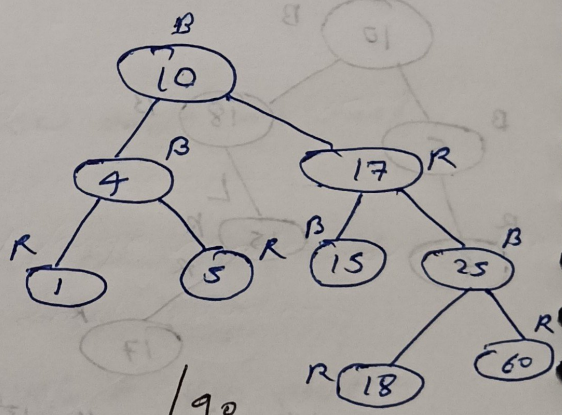
1



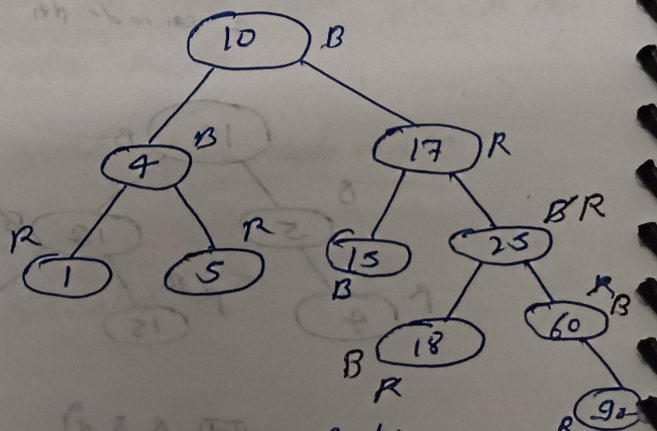
same as above

चाचा गायब दिक्कत है

Right rotation & exchange color

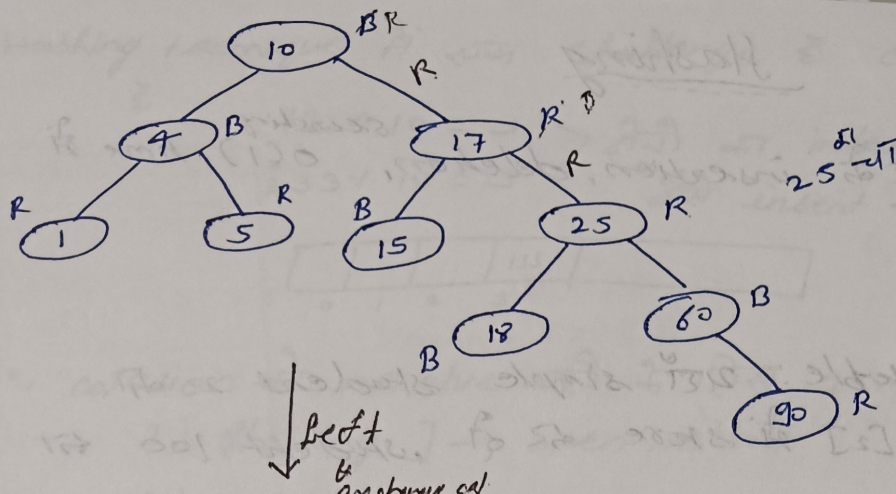


go

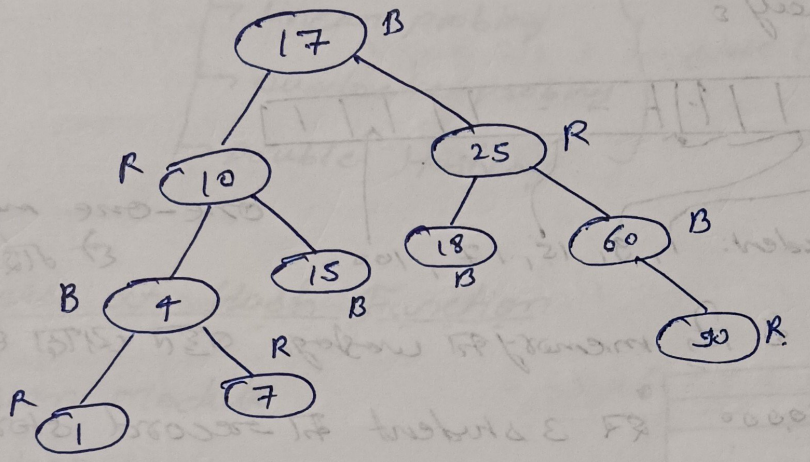


15 का color Red हो गया अब इसको newly inserted node मान लेंगे

uncle Red है



25 का चाचा Black है तो दिक्कत है



Important :-

1) चाचा का color देखना है Red है तो चाचा और पापा का color change कर दो, दादा root node नहीं है तब दादा का भी color change कर दो color change करने से problem हो रही है तो फिर से condition 1 या condition 2 hold कर रही होगी।

2) चाचा का color Black या null है तो newly node से पापा, दादा तक जाओ (सगे दादा के पास) वहाँ से newly node तक आओ पता चल जाएगा LL, RR, LR, RL क्या है, LL, RR में simply rotate कर दो जो node rotate हो रहे उनका color exchange कर दो

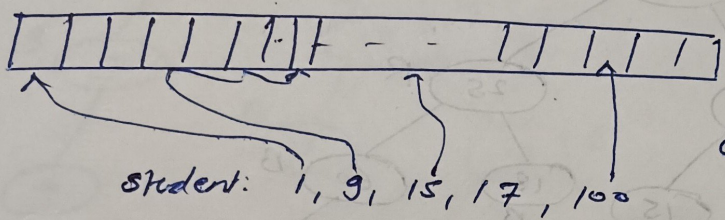
LR, RL second rotation में जो node rotate हो रहे उनका color exchange कर दो।

Hashing

Hashing का use कर के insertion, deletion, searching O(1) time में कर सकते है।

Direct Address table: यहाँ simple student 2 का record A[2] में store कर दो, student 100 का A[100]

A - array है



student: 1, 9, 15, 17, 100

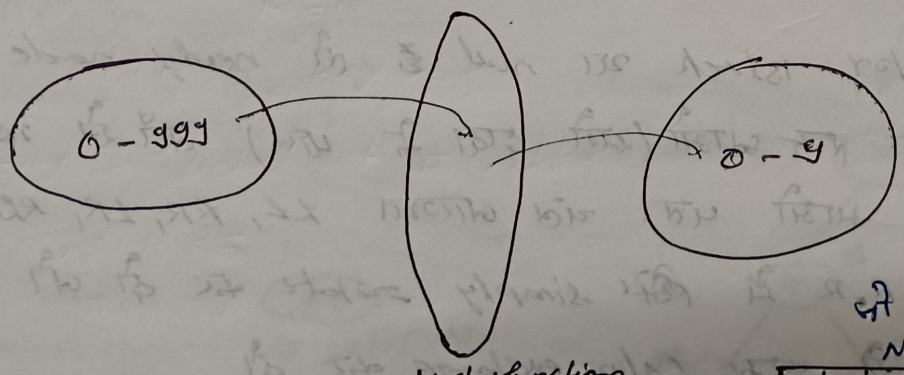
one-one mapping हो गई

इसका drawback है कि memory का wastage बहुत ज्यादा है।

student 1, 10, 10,000 इन 3 student का record store करने के लिए array का size 10,000 चाहिए।

Hashing:- बड़ी value को छोटे में map करेंगे छोटे value में।

Hashing:- Taking a set of very large number and then mapping into a set of small number



जो space खाली है उसमें NULL मार देंगे।

990	N	N	N	N	N	146	N	N	N
0	1	2	3	4	5	6	7	8	9

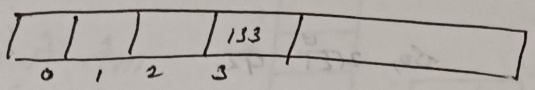
146 % 10 = 6 So, 146 index = 6

990 % 10 = 0 पर store होगा।

Hashing technique में एक challenge है collision हो सकता है

1334 % 10 = 3
2334 % 10 = 3

> दोनों का index 3 आया अब कैसे insert करेंगे।



So, collision को resolve करने के लिए 2 method है

1) Chaining] → open Hashing
2) Open Addressing
   - linear probing
   - Quadratic probing
   - Double Hashing

} close Hashing

Types of Hash Function

1) Division Modulo

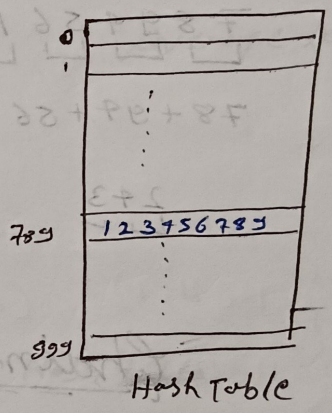
M = 1000

key = 123456789

HF(key) = key Mod M

= 123456789 mod 1000

= 789



2) Mid square (पहले square calculate करेंगे फिर mid)

Mid कितने digit लेंगे अगर M ⇒ n digit का है तो Mid ⇒ n-1 digit की

M = 1000 ⇒ 4 digit

key = 121

$(121)^2$ = 14641

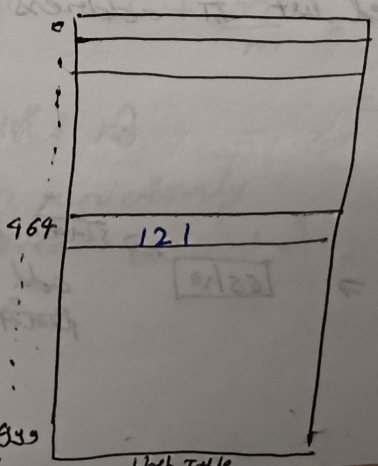
Mid ⇒ 3 digit

* M = 100 होती तब

14641

46 64

Mid इन दोनों में से कोई भी ले सकते है।



3.) Digit extraction

M = 1000

Key = 789 456 123

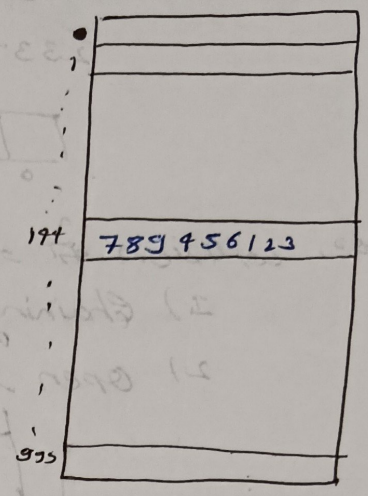
digit को कैसे pick करेंगे

M = n digit तो

Then

Key digit pick करेंगे = (n-1) digit

So, यहाँ पर 3 digit extract करेंगे।



789 + 456 + 123

= 1368 ⇒ 4 digit है अब digit extraction में 3 digit ही लेंगे।

= 136 + 8

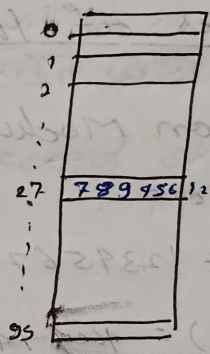
= 144

Suppose इसी में M = 100 होती तब

7 8 9 4 5 6 1 2 3

78 + 94 + 56 + 12 + 3

243 24 + 3 = 27



Chaining (open Hashing)

keys: 25, 97, 80, 65, 38, 40, 59, 75, 17, 99, 35, 23

M = 10

h(key) = key mod M

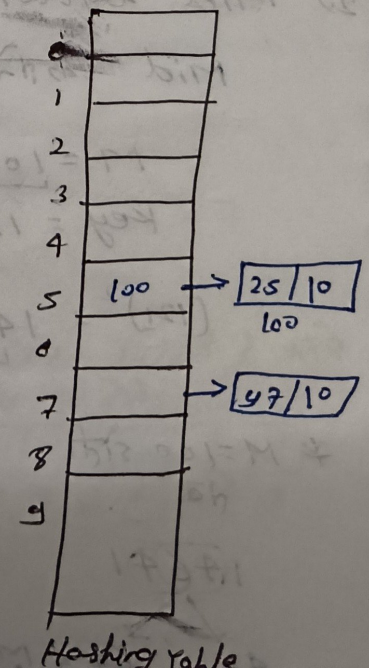
Array में linked list का address store होता है।

25 % 10 = 5

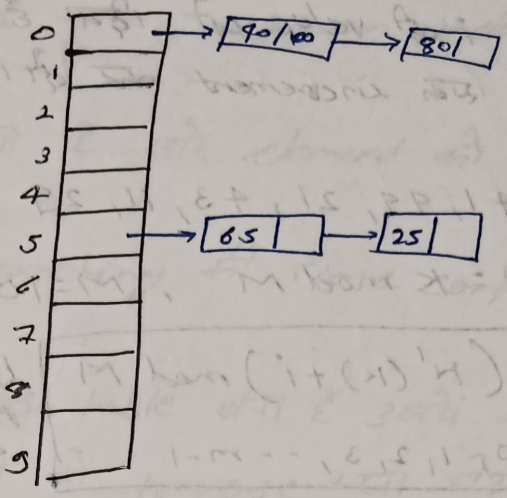
97 % 10 = 7

⋮

65 % 10 = 5 ⇒ 65|10 ⇒ इसको beginning में add करेंगे जिससे time complexity O(1)



यहाँ हर index के corresponding linked list होगा



ऐसे ही linked list create होगा
हमेशा beginning में होगा

Drawback :-

→ यहाँ collision तो resolve हो जाता है पर extra space लग रहा है।

→ अगर key = 10, 20, 30, 90, 50, ... तो index 0 पर सारी value store हो जाएगी। यहाँ पर searching में time complexity O(n) हो जाएगी worst case में.

→ Chaining में

Insertion - O(1)
deletion - O(n) worst case
searching - O(n) worst case

Average length = load factor के बराबर होगा।

n = no. of element filled in Hash table.

$$\text{Load}(\alpha) = \text{load factor} = \frac{n}{m}$$

⇒ हर index को equal no. of element मिला।

→ Hash table में element uniformly distributed हैं तब searching में Time लगेगा = $C + \frac{n}{m} = C + \frac{C \cdot m}{m}$

$= O(1)$

Assumption यहाँ है कि Hash function uniformly distribute करेगा element को।

→ Linear Probing in Hashing

simple initially i की value 0 दे दिया है और collision occur होता है i को एक increment कर देंगे।

keys: 56, 79, 41, 99, 21, 43, 11, 29

$H'(K) = K \bmod M$, $M = 10$

$$H(K,i) = (H'(K) + i) \bmod M \quad \text{for } i = 0, 1, 2, 3, \ldots m-1$$

Linear probing की formula

0	
1	41
2	21
3	43
4	44
5	11
6	56
7	
8	
9	79

$H(56,0) = (56+10,0) \% 10 = 6$

$H(79,0) = 9$

$H(41,0) = 1$

$H(99,0) = 9$ collision occur किया, अब i का value increment कर दी

$H(21,0) = 1$ collision occur किया }, अब i का value increment कर दी

$H(21,1) = (1+1) \% 10 = 2$ ⇒ खाली है तो यहीं डाल देंगे

$H(43,0) = 3$

$H(11,0) = 1$ collision occur

$H(11,1) = (1+1) \% 10 = 2$ collision

$H(11,2) = 3$ collision

$H(11,3) = 4$ collision

$H(11,4) = (1+4) \% 10 = 5$ ⇒ खाली है यहाँ insert कर देंगे।

$H(29,0) = 9$ collision }.

Conclusion से निकला linear probing में अगर space खाली नहीं है तो उसके next location में check करते हैं।

इसमें No. of collision फूँक सकता है।

यहाँ पर No. of collision = 6 है

Drawback: 11 को check करना है 1 पर check किया नहीं है, so next पर check करेगी नहीं है --- so on index 5 पर मिलेगा।

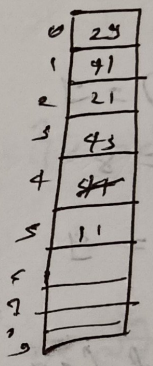
worst case में हो सकता है सारे element को check करना पड़े।

worst case में searching की Time complexity - O(n)

→ suppose previous जो Hash table बना है उसमें 31 को insert करना हो तब 6 element को traverse करना होगा। index 7 खाली मिलेगा इसको बोलते हैं primary clustering

→ deletion is not recommended in linear probing

suppose 44 delete कर दिए
अब 11 को search करेंगे, जब search करेंगे तब 43 के बाद रुक जाएंगे, so 11 नहीं present है ऐसा output आएगा।



→ shift कर दे इसका एक solution है पर हमेशा shifting work नहीं करेगा।

table 21 delete कर दिए अब 43 को index 2 में नहीं shift कर सकते so, इसका solution है delete करने के बाद सारे element कि फिर से Re-Hashing कर दो

another solution is जो element delete कर रहे हैं उसके जगह कोई special symbol रख दे जिससे पता चलेगा कि element delete हो गया।

→ shifting work करेगा जब तक 44 को delete कर दिए अब 11 को index 4 पर रख दिए because 44 नहीं होगा तो 11 वहीं आता।

Advantage: space कम ले रहा है Chaining से।

→ Quadratic Probing

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod M$

$h'(k) = k \bmod M$ $\quad c_2 \neq 0$

i. $i = 0, 1, 2 \cdots (M-1)$

c_1, c_2 की value दी होती है नहीं दी है तो assume

$c_1 = c_2 = 1$

key: 89, 18, 49, 58, 69, 29

$M = 10$

$h(89,0) = 9$

$h(18,0) = 8$

$h(49,0) = 9$ collision

अब $i = 1$

$h(49,1) = (9+1+1) \% 10$

$= 11 \% 10$

$= 1$

$h(58,0) = 8$ collision

$h(58,1) = (8+1+1) \% 10$

$= 0$

$h(69,0) = 9$ collision

$h(69,1) = (9+1+1) \% 10$

$= 1$ collision

$h(69,2) = (9+2+4) \% 10$

$= 15 \% 10 = 5$

$h(29,0) =$ collision

$h(29,1) =$ collision

$h(29,2) =$ collision

$h(29,3) = (9+3+9) \% 10$

$= 21 \% 10 = 1$ collision

$h(29,4) = (9+4+16) \% 10$

$29 \% 10 = 9$ collision

$h(29,5) = (9+5+25) \% 10$

$39 \% 10$

9 collision

$h(29,6) = (9+6+36) \% 10$

$= 51 \% 10$

1 collision

0	58
1	49
2	
3	
4	
5	69
6	
7	
8	18
9	89

$h(29,7) = (9+7+49) \% 10 = 5$ collision

$h(29,8) = (9+8+64) \% 10 = 1$ collision

$h(29,9) = (9+9+81) \% 10 = 9$ collision

अब stop कर जाएंगे और 29 को store नहीं करेंगे।

Linear Probing
Primary clustering
Secondary clustering

Quadratic Probing
Secondary clustering

→ कुछ time बाद space मिल जाता।

Secondary clustering
exactly same prob आना है दो key में
above code में $29 \Rightarrow 9, 1, 5, 1, 9, 9, 1, 5, 1, 9$ } same आया
$19 \Rightarrow 9, 1, 5, 1, 9, 9, 1, 5, 1, 9$
और 19 insert नहीं कर पाया

→ Linear probing में space खाली है तो insert कर लेते हैं but quadratic probing में space खाली होते हुए भी insert नहीं कर पाते।

→ Quadratic probing में भी deletion नहीं करते

searching time worst case में $O(n)$ होता है

insertion - $O(1)$

Double Hashing

$$h(k,i) = (h_1(k) + i\,h_2(k)) \bmod M$$

$i = 0, 1, 2, \ldots (m-1)$

$h_1(k) = k \bmod M$, $h_2(k) = 1 + k \bmod (m-2)$

$k = 0, 1, 2, \ldots (m-1)$

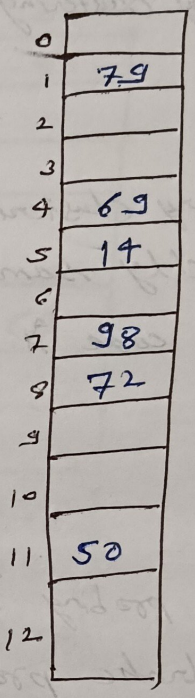
Double Hashing की बात 2 condition change भी हो सकता है.

79, 69, 98, 72, 14, 50 $\quad M = 13$

$M = 13$

$h_1(k) = k \,\%\, 13$

$h_2(k) = 1 + k \,\%\, 11$



$h(79,0) = 79 \,\%\, 13 = 1$

$h(69,0) = 69 \,\%\, 13 = 4$

$h(98,0) = 98 \,\%\, 13 = 7$

$h(72,0) = 72 \,\%\, 13 = 7$ collision

$h(72,1) = (7 + [1 + 72 \,\%\, 11]) \,\%\, 13$

$= (7 + 7) \,\%\, 13$

$= 14 \,\%\, 13$

1 collision

$h(72,2) = (7 + 2\,[1 + 72 \,\%\, 11]) \,\%\, 13$

$= (7 + 14) \,\%\, 13$

$21 \,\%\, 13 = 8$ free है

$h(14,0) = 14 \,\%\, 13 = 1$ collision

$h(14,1) = (1 + [1 + 14 \,\%\, 11]) \,\%\, 13$

$= (1 + 4) \,\%\, 13$

$= 5$ free है

$h(50,0) = 50 \,\%\, 13 = 11$

Drawback linear probing → quadratic probing → double hashing

	LP	QP	DH
primary clustering	✓	×	×
secondary clustering	✓	✓	×

Time complexity

worst case searching = $O(n)$

→ यहाँ भी deletion नहीं perform करते.

Q Consider the following input

4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199

hash function is K mod 10

4322 % 10 = 2

1334 % 10 = 4

1471 % 10 = 1

9679 % 10 = 9] same

1989 % 10 = 9] same

same

6171 % 10 = 1

6173 % 10 = 3

4199 % 10 = 9

1) 9679, 1989, 4199 hash to same value

2) 1471, 6171 hash to the same value

3) All element hash to the same value

4) Each element hashes to a different value

Q $h(K) = K \bmod 10$

CRT = Linear Probing

For given hash table.

How many different insertion sequence are possible?

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

42, 23, 34, 52, 46, 33

↓ इन 3 को चाहे जैसे place करे यहाँ पर answer.

↓ exact position पर नहीं है

↘ exact position पर नहीं है

3! आएगा keeping 52, 46, 33 same position

46 अपनी exact position है अब check करते है उसको नहीं रख सकते

_ 42 _ 23 _ 34 _ 52 _ ■ _ 33 ✗ ↳ यहाँ बस problem होगा।

(5)

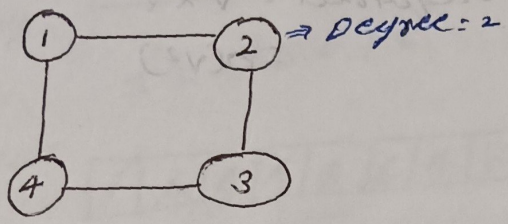
No. of possible way : 6 × 5 = 30

बाकी हर जगह रख सकते है

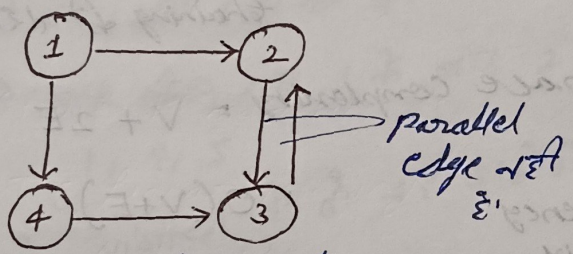
Graph

↳ set of vertices & edges

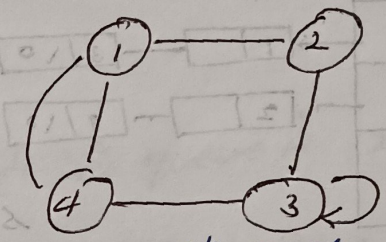
Degree:- उस vertex के कितने adjacent vertex हैं वही degree है.



→ ऐसा graph जिसमें self loop या parallel edge present हो उसे multigraph बोलते हैं, दोनों में से कोई present ना हो तब simple graph बोलते हैं.



simple Graph



multi-graph

* Path 1-2-3-4-1

1-2-3-4-1 ⇒ ये cycle है

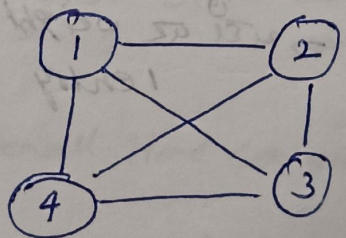
2-1-3-1-2 ⇒ cycle नहीं है

For a cycle starting & ending vertex must be be same & between them all vertex must be distinct.

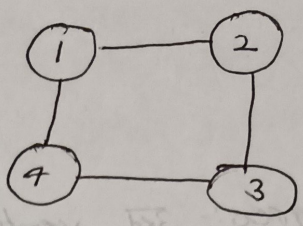
2-3-2 ⇒ cycle है

→ complete Graph

↳ Edge = $\frac{V(V-1)}{2}$ सारे vertex (या node) connected होते है.



→ Representation of Graph

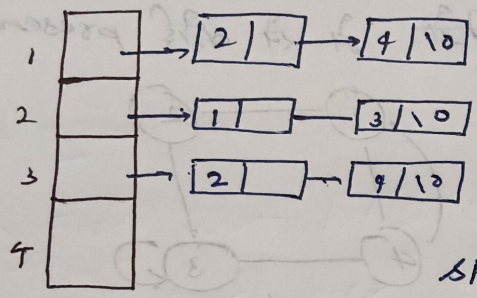


1) Adjacency Matrix

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

space required = $V \times V$
$= O(V^2)$

2) Adjacency List



node से जो भी connected उसको connect कर दो उस index में chaining की तरह से।

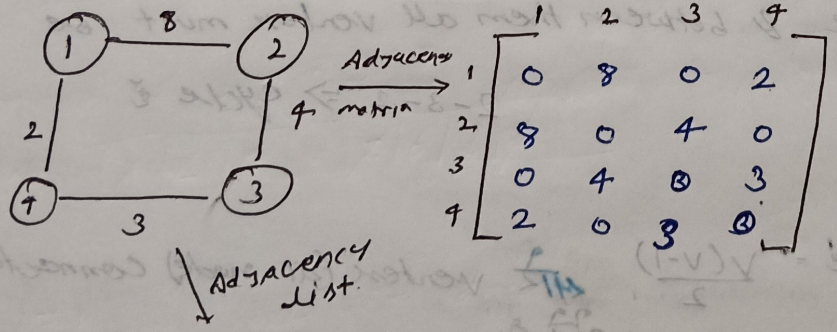
space complexity = $V + 2E$
$= O(V+E)$

Graph → Dense - Adjacency matrix
→ Sparse - Adjacency List

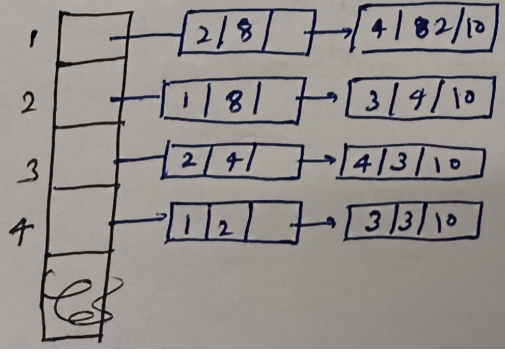
⇓

no. of edges कम हो तो sparse graph

Dense :- no. of edge ज्यादा हो तो Dense graph.



यहाँ weight लिख देते हैं

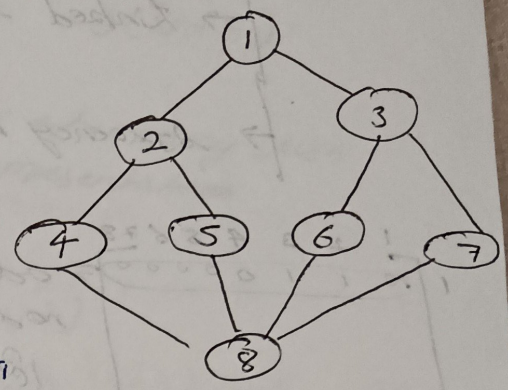


⇒ यहाँ पर weight के लिए 1 entry बढ़ा देते हैं।

→ Breadth First Search (BFS)

⇓

visited node का order unique नहीं होता



visited array

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

→ O(V)

1 2 3 4 5 6 7 8

queue

1	2	3	4	5	6	7	8

↑

इसमें adjacent node को डालेंगे जैसे 1 को

visit किए 1 queue में गया अब 1 का adjacent देखेंगे, अगर वो visited नहीं है तब उनको queue में डाल देंगे

1 हो गया अब pop कर दो

2 को pick किए अब 1 के बाद 2 ही है queue में 2 का adjacent

1, 4, 5

⇓

2 visited है तो 4 ही डालेंगे। जरूरी नहीं 4 पहले डालेंगे

फिर 5 order change कर सकते है इसलिए ये unique नहीं है।

u = 3 { 6, 7, 1 } (✓ ✓ ×)

u = 4 { 2, 8 } (× ✓)

u = 5 { 2, 8 } (× ×)

u = 6 { 3, 8 } (× ×)

u = 7 { 3, 8 } (× ×)

u = 8 { 4, 5, 6, 7 } (× × × ×)

visited array की time complexity = O(V)

सारे node को queue में डाल रहे queue में insertion deletion की time complexity O(V)

Fir check कर रहे है visited array में की adjacency list उसमें ~~search करने में time~~ 2E × O(1)

adjacent node पता करने में: O(E)

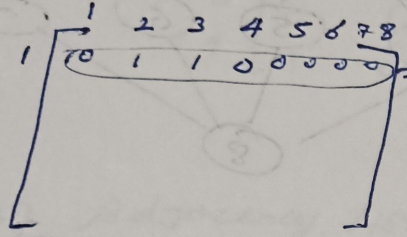
So, overall time complexity is O(V+E)

space complexity = O(V)

BFS

Graph की representation

- Linked list → time complexity $O(V+E)$, space complexity $O(V)$
- Adjacency matrix → time complexity $O(V^2)$, space complexity $O(V)$



adjacent node पता करने के लिए पूरा array scan करना पड़ेगा।

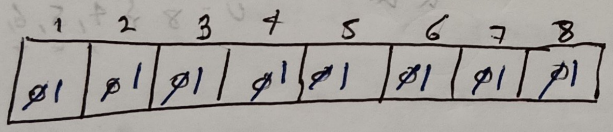
$V * V = V^2$

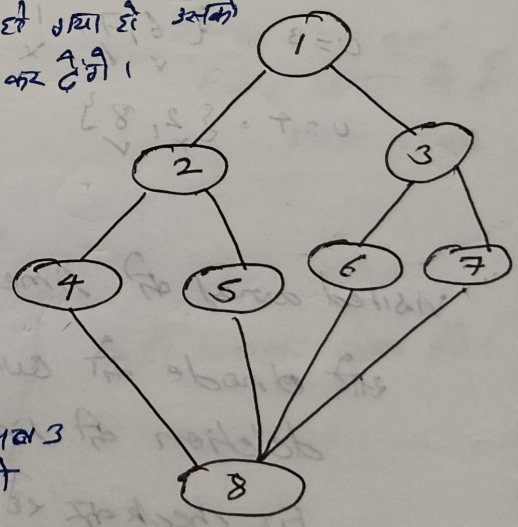
No. of vertex — कि vertex का adjacent कौन adjacent है

→ Depth First search (DFS)

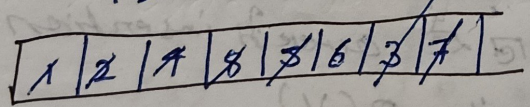
यहाँ पर stack use करेंगे वहाँ पर queue use किए थे।

सब कुछ same as previous बस इसमें जैसे ही new node मिला इसका adjacent node को stack में डाल देंगे और जिसका सारा adjacent node cover हो गया है उसको pop कर देंगे।





1 2 4 8 5 6 3 7



अब 3 को

u = 1 → new node इसका adjacent

{2, 3}

↓ new अब इसका adjacent

{4, 5}

↓

{2, 8}

{4, 5, 6, 7}

{3, 7} pop

अब 8 के पास

{3, 8}

{1, 7}

{2, 1, 8} 5 को pop कर दिया

order unique नहीं होता। यहाँ भी

→ Time complexity is same as BFS $O(V+E)$

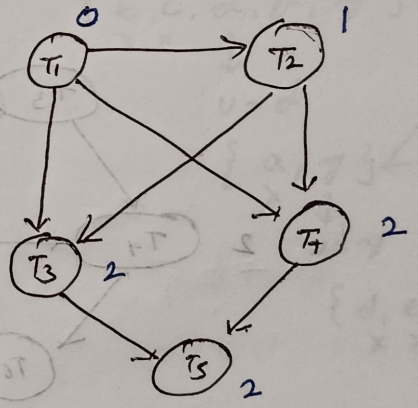
Space complexity $O(V)$ → For adjacency list representation

For adjacency matrix
- Time complexity $O(V^2)$
- Space complexity $O(V)$

→ Topological Sort:-

↓

Given graph must be Directed acyclic graph.



Vertex ⇒ Task

Edges ⇒ dependency

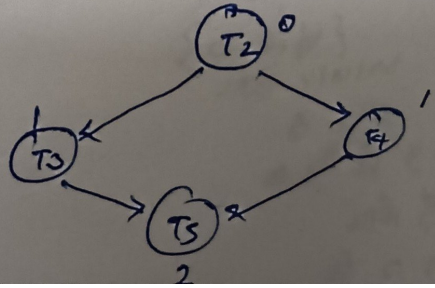
Indegree = No. of coming node to that node.

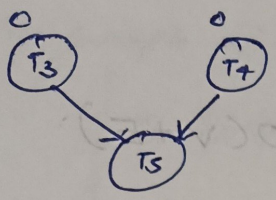
1st step ⇒ सारे node की indegree निकालेंगे। जिसकी indegree zero होगी उसे लिख लेंगे।

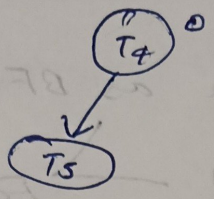
(T1)

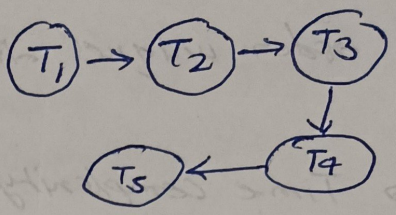
2nd step ⇒ 0 indegree वाले node को delete कर के। फिर graph बनाएंगे और उसमें 0 indegree वाला खोजेंगे

किसी दो का indegree zero आए तो किसी को भी pick कर सकते।









So, order of topological sort

$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5$

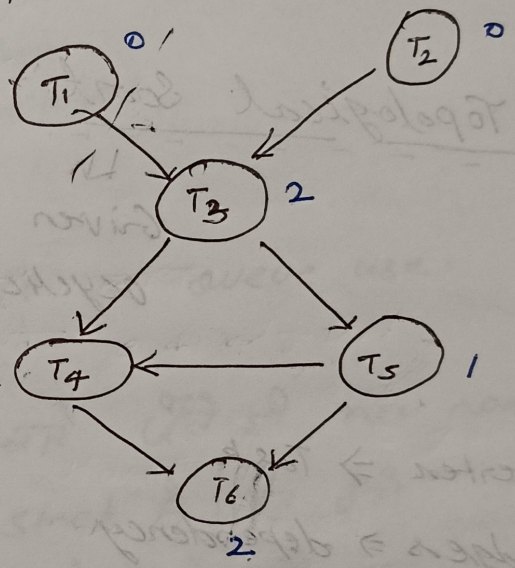
मतलब, Topological sort का sequence unique नहीं होता।

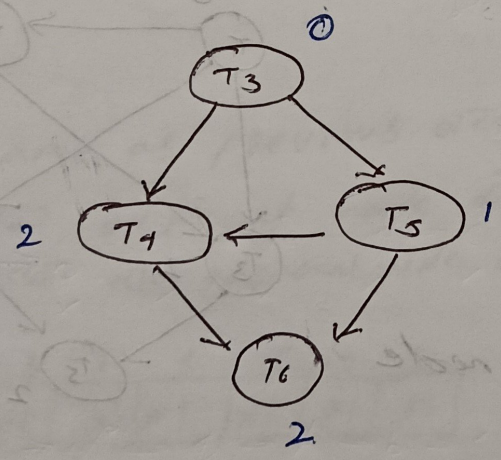
Q Topological sort का sequence निकालो

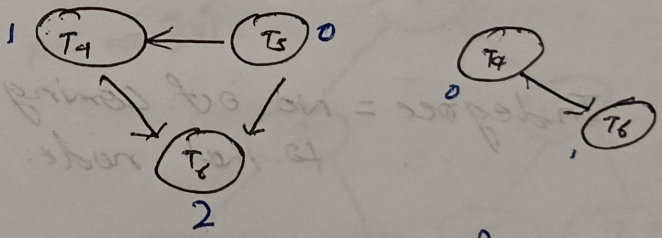
$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_5 \rightarrow T_4 \rightarrow T_6$

or

$T_2 \rightarrow T_1 \rightarrow T_3 \rightarrow T_5 \rightarrow T_4 \rightarrow T_6$







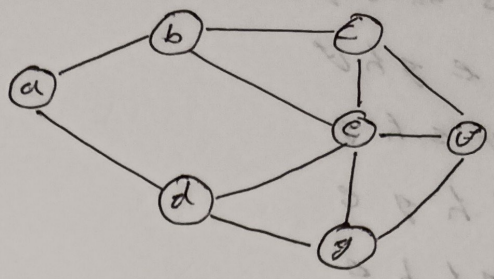
→ Graph में cycle होगा तो उसका Topological sort नहीं निकाल सकते।

* DFS ज्यादा important है

DFS का sequence लिखना

~~not possible~~

a b c e d g f



option दिया है check करना है तो simple हर node का adjacent check करो अगर उसमे से कोई present है तो यही है

1.) a b e f d g c ✗

v = a
{b, d}
✓

v = b
{c, e, a}
✓

v = e
{b, c, f, g, d}
✓

v = f
{c, g}
✗

because f के बाद d है but f के बाद c या g में से कोई होना चाहिए था

2.) a b e f c g d ✓

v = a
{b, d}
✓

v = b
{c, e, a}
✓

v = e
{b, c, f, g, d}
✓

v = f
{c, g}
✓

v = c → pop
{b, e, f}
→ सारे visited है pop

अब यहाँ आए गए
g मिला सही है

v = g
{d, e, f}
✓

v = a
{b, d}
✓

v = b
{a, c, e}
✗

v = c
{b, e, f}
✗

v = e
{b, c, d, f, g}
✗ ✗

f मिला f visited है
g मिला g भी visited

v = d
{a, g}
✗

v = g
{d, e, f}
✗ ✗

v = f
{c, g}
✗ ✗

3) a d g e b c f

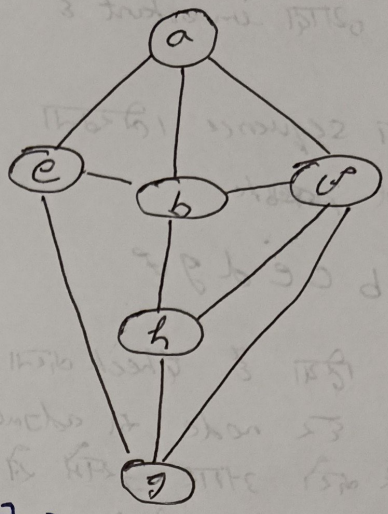
* a d b c g e f

a → b → e → f → c → g → d
✓ ✓ ✓

{e, f, b}
अब visited है अब e f से e पर आके देखेंगे की c तक जाता या है कि नहीं

Q DFS का sequence कौन कौन सा है

1. ✓ a b e g h f
2. ✗ a b f e h g
3. ✗ a b f h g e
4. ✓ a f g h b e



Direct नहीं कर सकते हैं

हम Check करना है
a से next node में direct जा सकते हैं
यहां नहीं

a → b → e → g → h → f
✓ ✓ ✓ ✓ ✓

a → b → f → e → h ✗
✓ ✓ ↳ direct नहीं जा सकता
अब देखेंगे e का adjacent क्या है
{a, g}
↳ g visited नहीं है

मतलब
नहीं जा सकते अब
e → h

a → b → f → h → g → e
✓ ✓ ✓ ✓ ✓

a → f → g → h → b → e
✓ ✓ ✓ ✓ ✓