RB TREES

① - A node is either Red or Black.

② - The root and leaves (NIL) are Black.

③ - If a node is Red, its children are Black.

④ - All paths from a node to its (NIL) descendants contain same number of black nodes.

[Remember: When counting no. of black nodes in this path, we ~~do~~ DO NOT count the starting node!]

- Each node has its own black-height which can be calculated using this

⇒ ~~XTRA NOTES~~

- Nodes req. 1 storage bit to keep track of color.
- The longest path (root to farthest NIL) is no more than TWICE the length of the shortest path (root to nearest NIL).

→ shortest path: All black nodes
→ Longest Path: Alternating b/w red & Black.

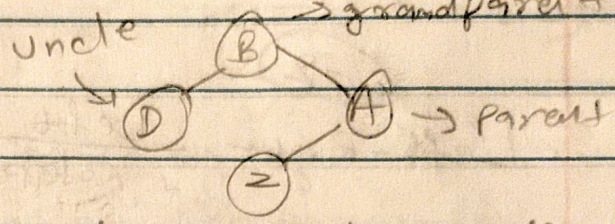
→ Operations | Insert, Delete, Search

↓ require rotation for violations

T.C.	O(logn) ← insert, delete, search
S.C	O(n) ← storing color of each node. Total nodes = 'n'

Case- For Insertions, we insert a node and color it 'red'. So that if it violates (2) or (3), its fairly easy to fix.

Z : node we insert



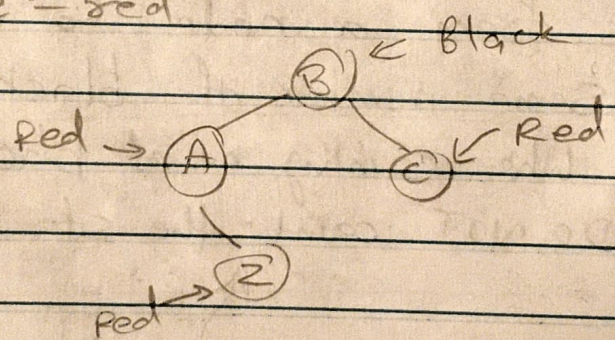
Case-1 Z's = root

(Z) ← Red (As we color node red while inserting)

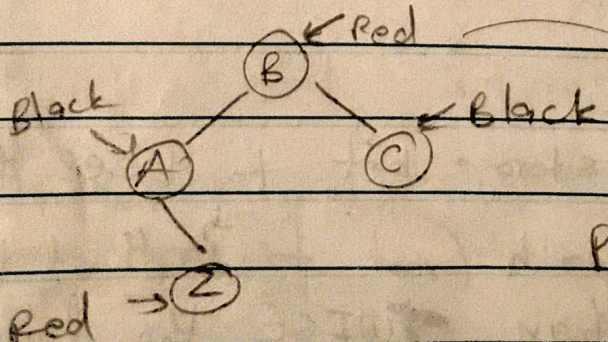
Fixing/Solution: Color the (Z) black, as root is always black

(Z) ← Black ✓

Case-2 Z.uncle = red

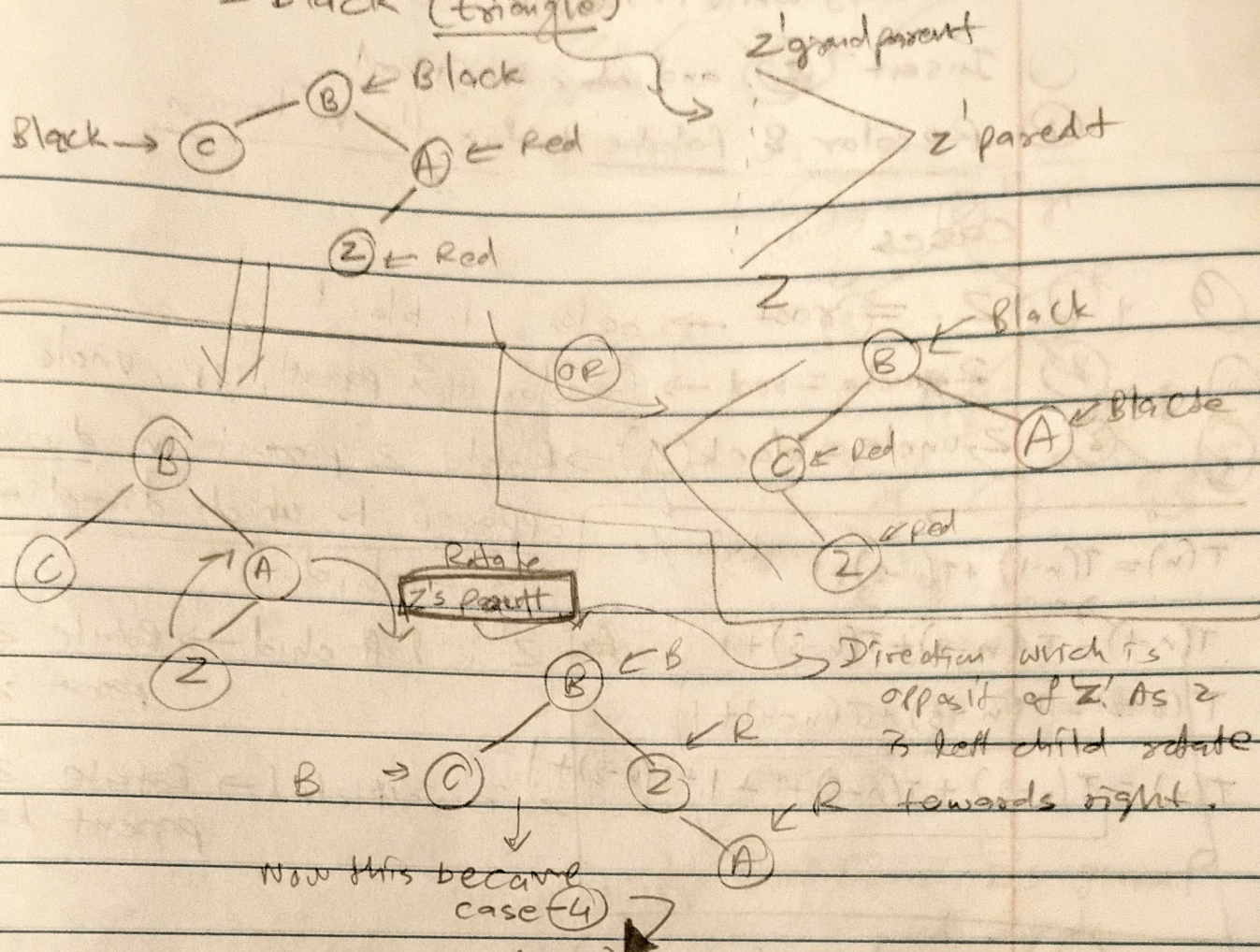


Fix/Solution: Recolor Z's parent, grandparent, uncle

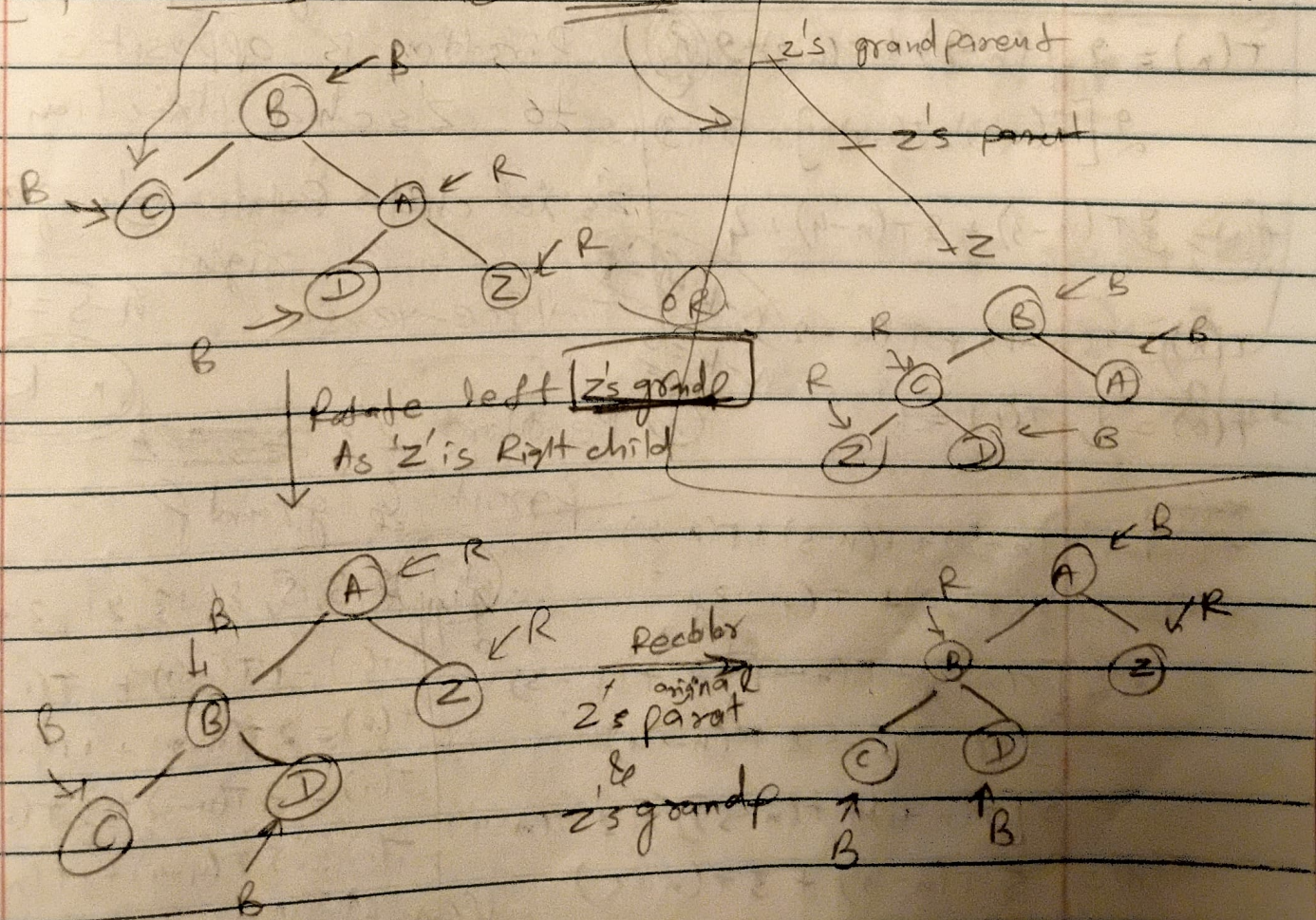


→ This (B) is not root, its part of a subtree.

Case-3 z.uncle = black (triangle)



Case-4. Z.uncle = black (line)



⇒ Strategy while Inserting

① Insert (z) and ~~ab~~ color it Red.

② Recolor & Rotate to fix the violation.

Cases

① z. = root → color it black

② z.uncle = red → Recolor it & z's parent, gp., uncle

③ z.uncle = black (Δ) → Rotate z' parent in direction opposite to which direction 'z' is a child.

Ex. z' is left child → Rotate z's parent right

z is right child → Rotate z's parent left

④ z.uncle = black (▽) → Rotate z's grandp similar

Direction is opposite to z's child direction

z's left child → Rotate z's grandp right

Vice-versa.

Recolor z's parent & grandp.

$T(n) = T(n-1) + T(n-2) + 1$ — triangle

$T(n-1) = T(n-2) + T(n-3) + 1$

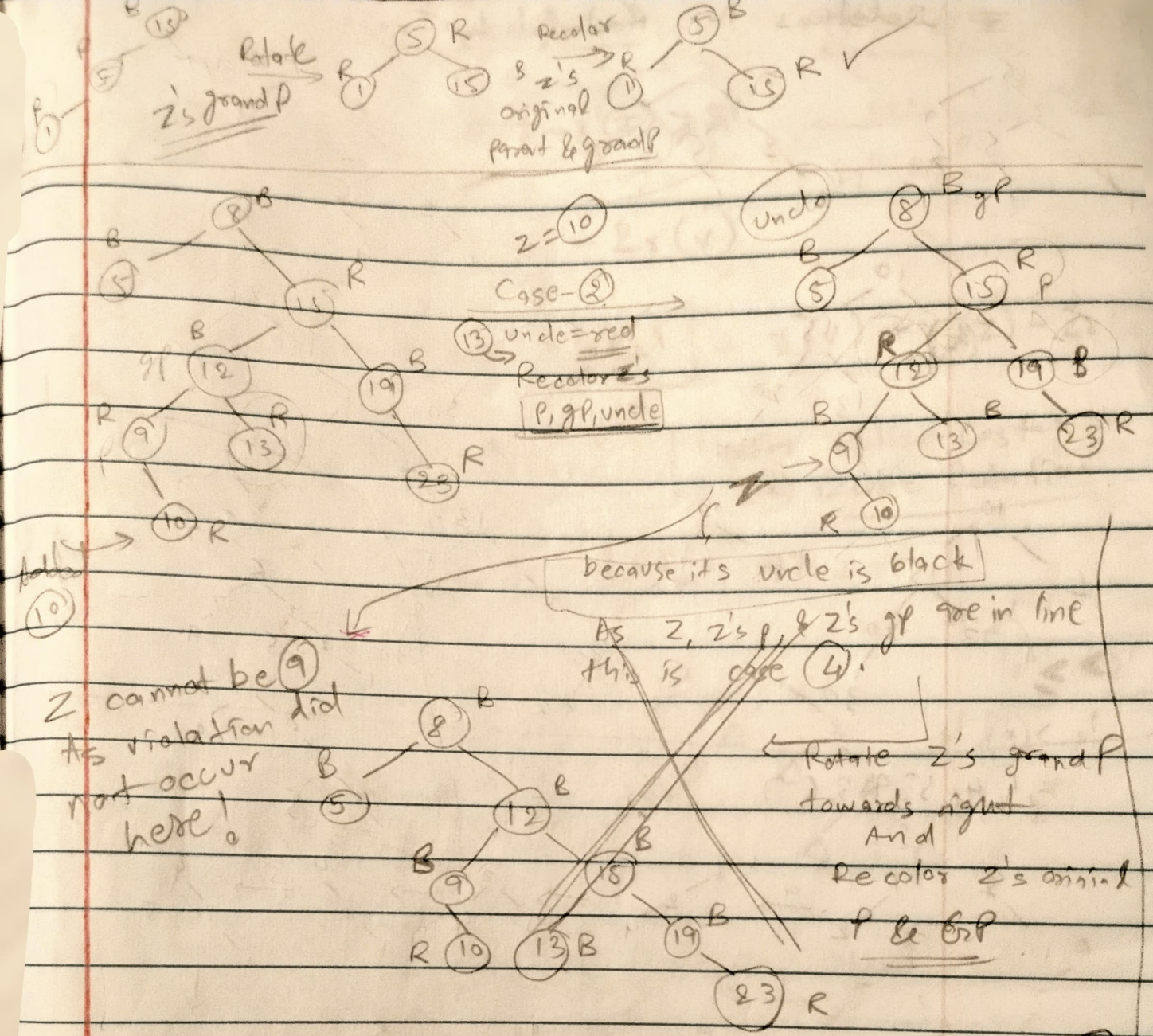
$T(n-2) = T(n-3) + T(n-4) + 1$

$T(n) = T(n-2) + T(n-3) + 1 + 1 + T(n-2) + 1$

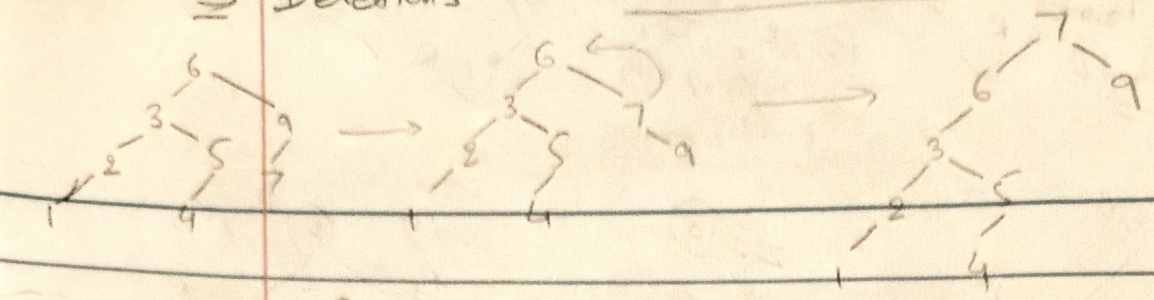
line

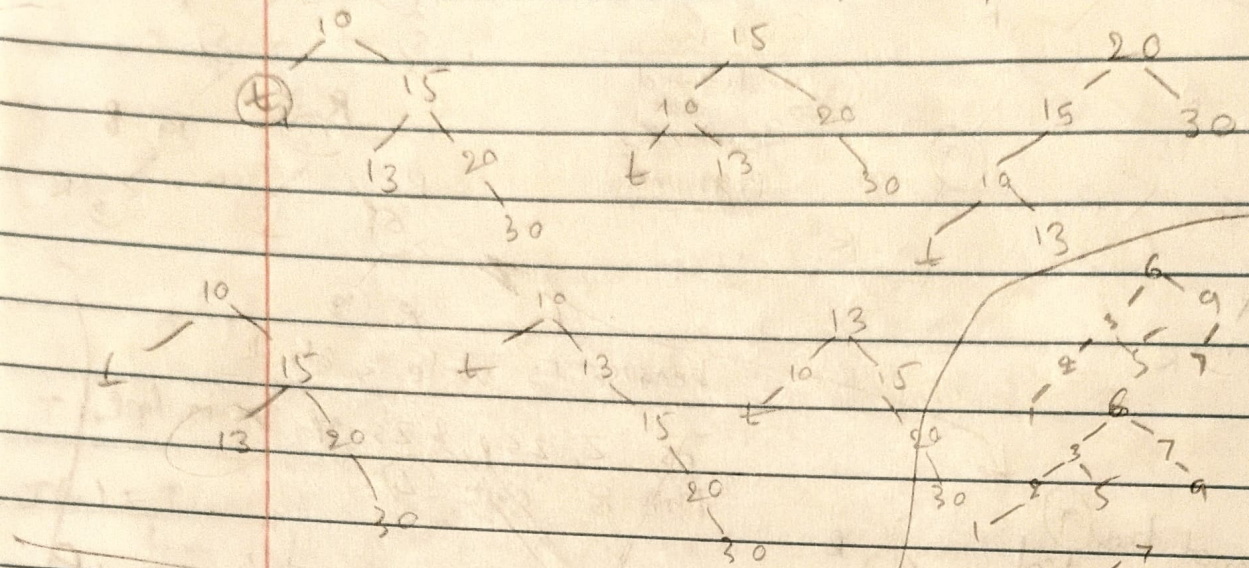
$n - \bar{5} = 0$

$n = k$

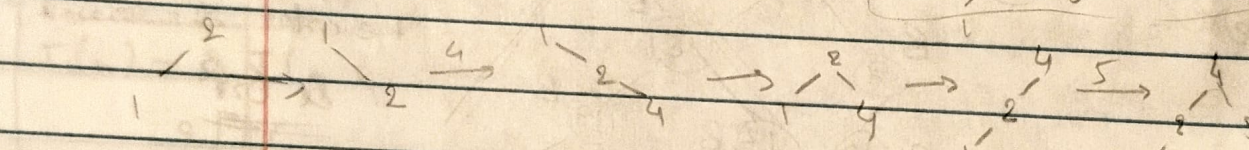

Rotate
z's grandP
Recolor
& z's original Parent & grandP
z = 10
Case-2
uncle=red
Recolor z's P, gP, uncle
uncle
gP
P
because its uncle is black
As z, z's P, & z's gP are in line this is case 4.
Rotate z's grandP towards right
And
Recolor z's original P & GrP
z cannot be 9 as violation did not occur here!
The node at which violation occurs becomes the z.
violation at 12 ← z
z' uncle 5 → is Black.
A ← Case 3
Rotate z' parent i.e. 15 towards right as z' is left child
violating node
z
uncle=Black
with (A)
Rotate left/Recolor gP

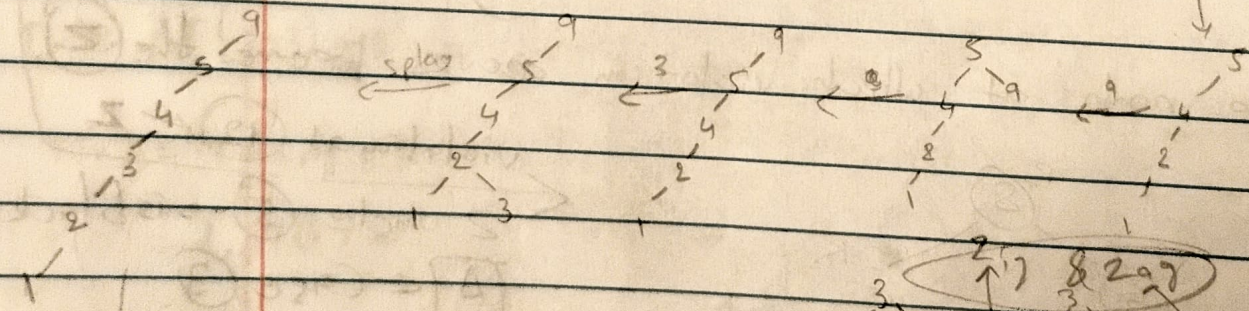
⇒ Deletions in Red-Black Trees





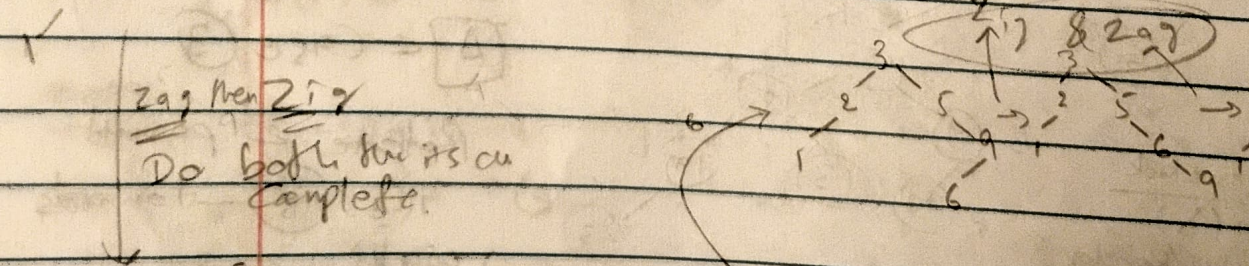
2, 1, 4, 5, 9, 3, 6, 7

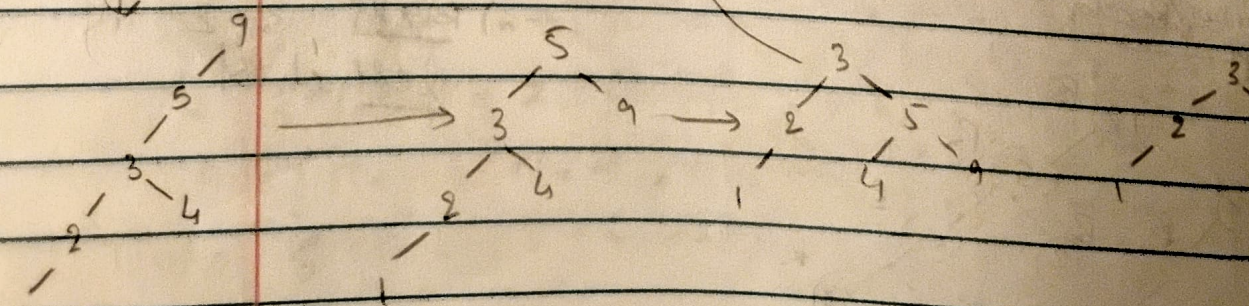




zag then zig

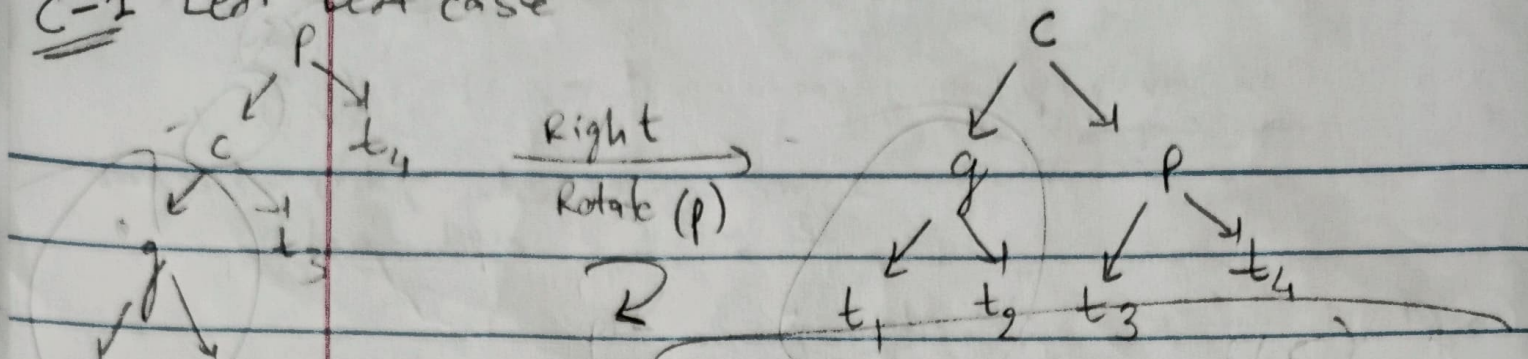
Do both the ifs as complete.



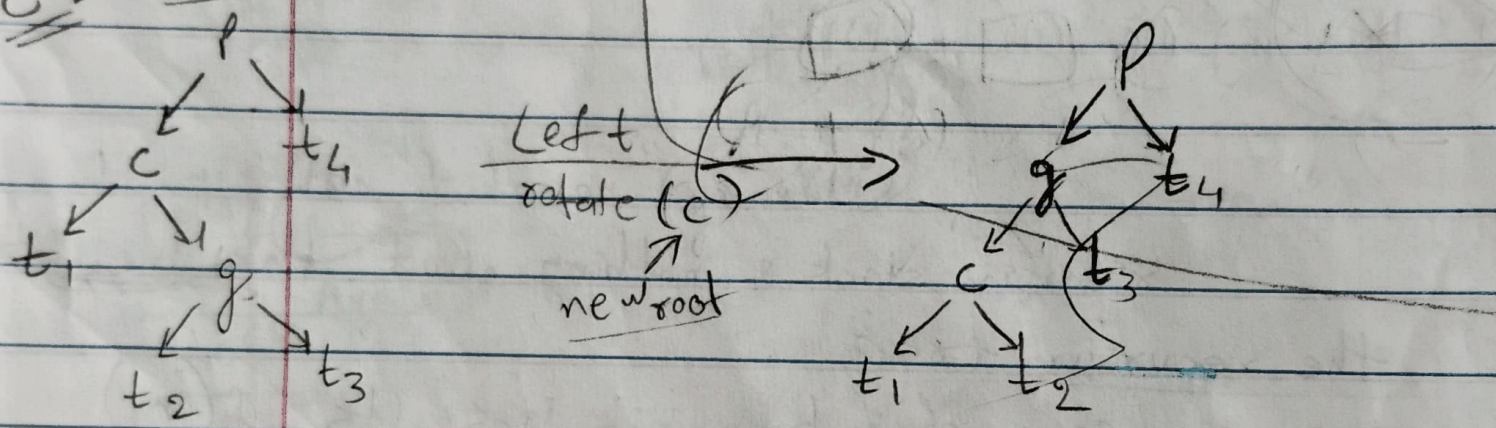


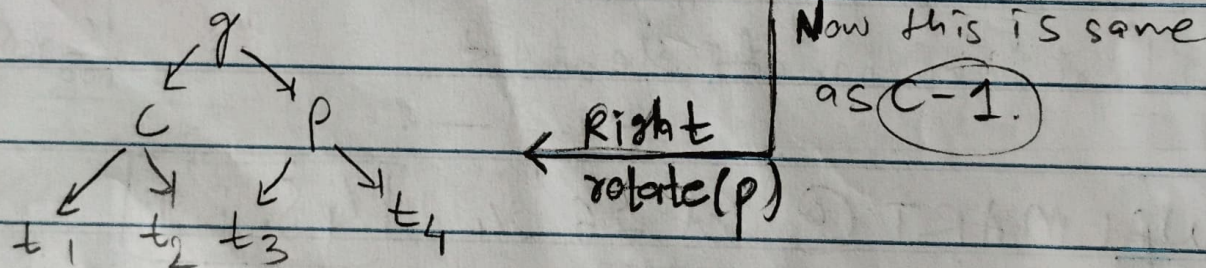
AVL TREES : Self-balancing Binary Search Tree

C-1 Left-Left case

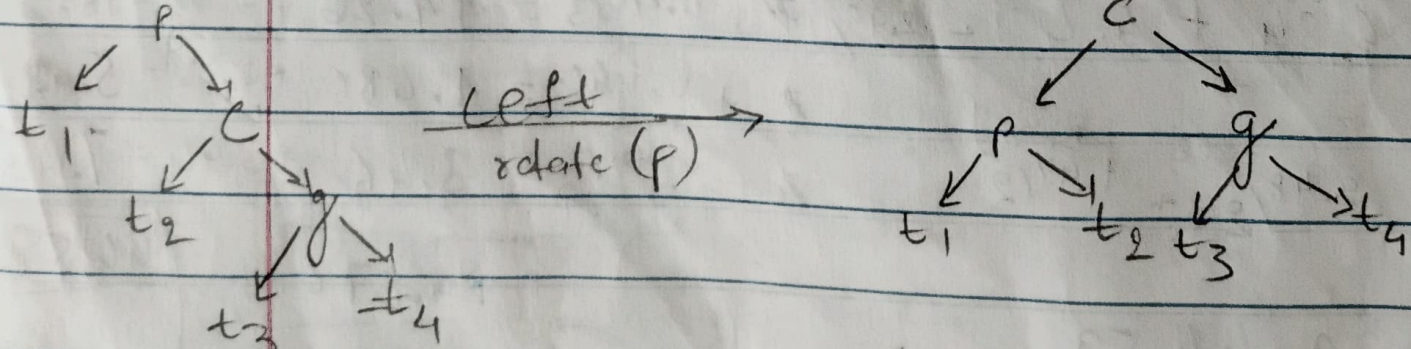


C-2 Left-Right case



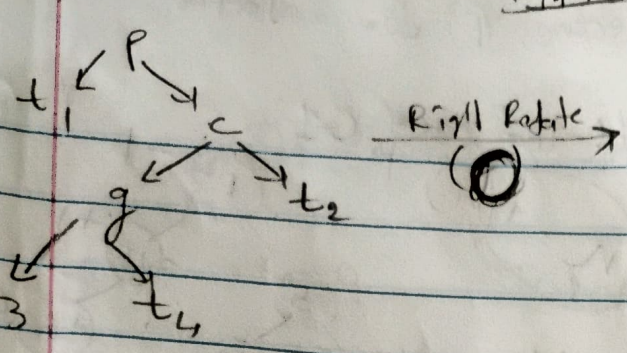


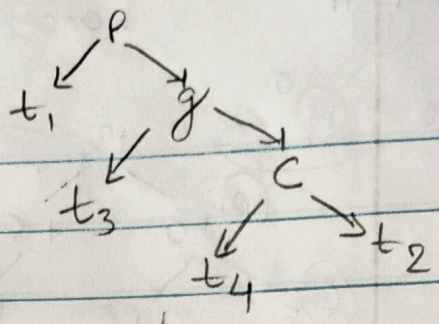
C-3 Right-Right case



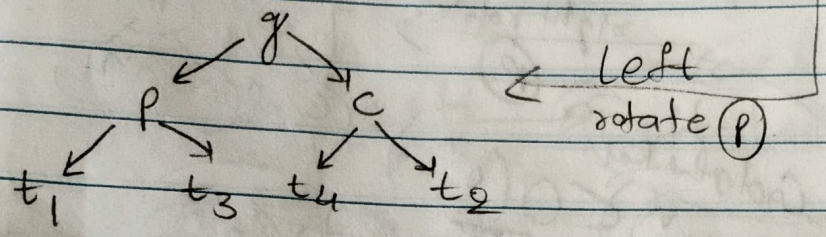
C-4 Right-Left case

726989
9472601





Same as C-3.



Aim: Is to get "p→c→g" in a single line, so that we can left OR right rotate over (p) and make the tree balanced!!.

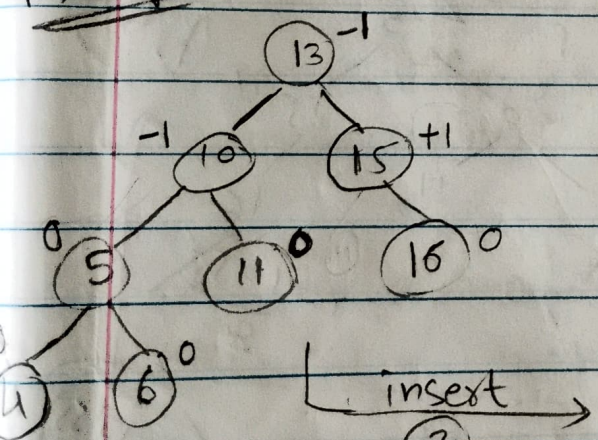
| p: Parent | c: child | g: grandchild |

Example

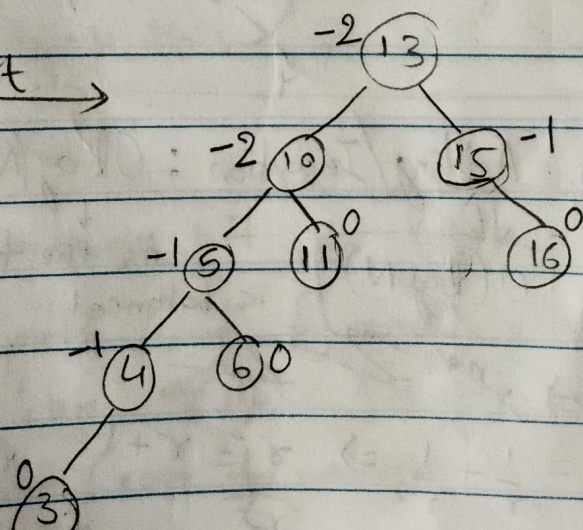
To check height balance/AVL Tree

= [height of right subtree − height of left subtree]

= {-1, 0, +1}



insert (3)



C-2

Left-Right $\xrightarrow[\text{Rotate}]{\text{Left}}$ Left-Left $\xrightarrow[\text{Rotate}]{\text{Right}}$ Balanced

C-1

Right-Left $\xrightarrow[\text{Rotate}]{\text{Right}}$ Right-Right $\xrightarrow[\text{Rotate}]{\text{Left}}$ Balanced

C-4

C-3