Trees

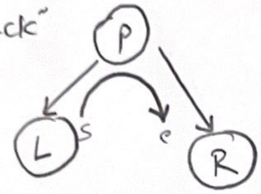
Structure:

```
Node {
    class data;
    int index;
    Node* left;
    Node* right;
    Node* parent;
}
```

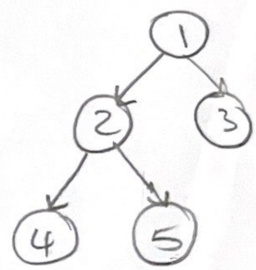
Traversals

DF (Depth) "stack"

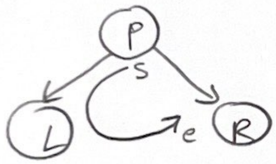
* In order
  LPR



4, 2, 5, 1, 3

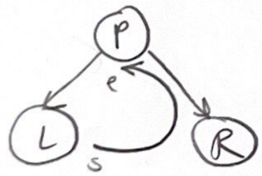


* Pre Order
  PLR



1, 2, 4, 5, 3

* Post Order
  LRP

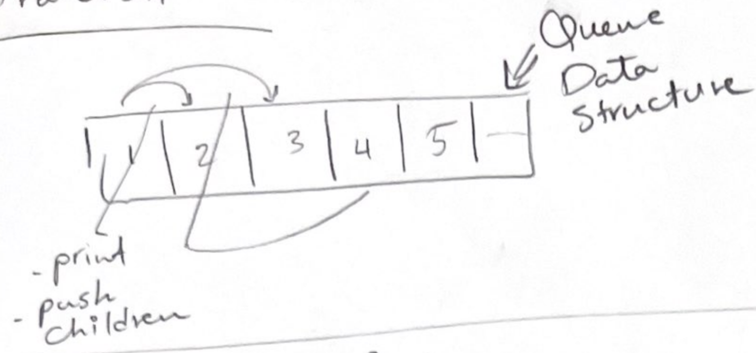


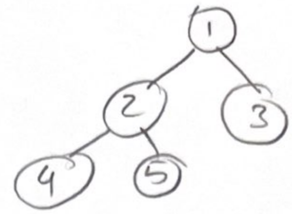
4, 5, 2, 3, 1

* Inorder Code:

```
traverse (left);
print;
traverse (right);
```

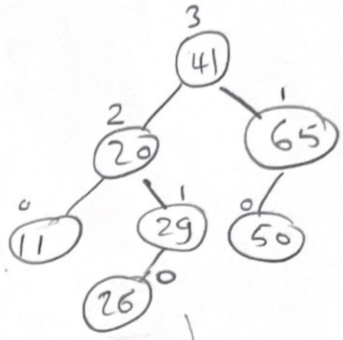
Breadth First Traversal



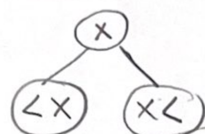


Calculate height of tree:

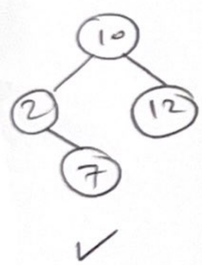
→ Post order traversal
* why? process children first
* maximize and + 1 ⇒ current height

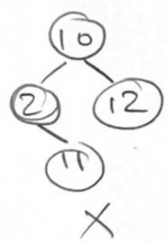


Binary Search tree



ex





in order traversal → sorted list
≡ Binary search tree

inorder : 11 20 26 29 41 50 65

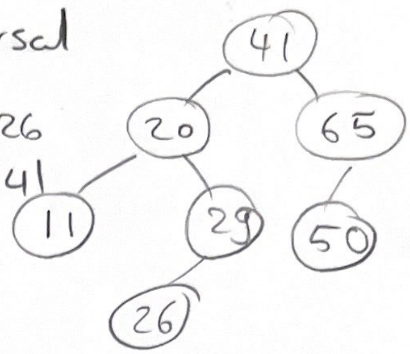
Search in BST

→ check if smaller go left
if larger go right
if equal then find

→ No backtracking, if not found in path will not be found in other

→ O(height) = O(log N)

* Successor



following number in inorder traversal
- → minimum in right subtree 20 → 26
- → parent where I'm left child 29 → 41
- → if no parent I'm left child ∴ max ⇒ 65

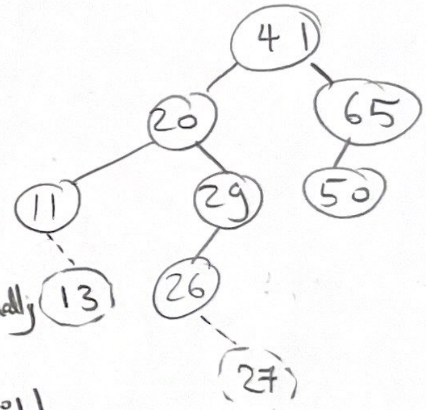
---

* Predecessor

Previous number in inorder traversal
- → max in left subtree 20 → 11, 41 → 29
- → parent where I'm right child 26 → 20

---

* Insert



→ Search till null, then place

* Delete

Case 1 : No children ⇒ delete normally
ex: 27

Case 2 : 1 child ⇒ delete then place child in place ex: 11

Case 3 : 2 children

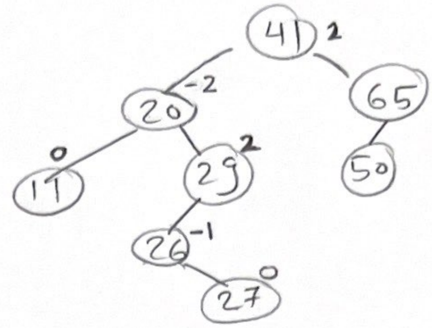
↳ Case 3a) replace successor instead of deleted if has no children delete 41 → replace it with 5

case 3b) successor has right children
↳ replace normally, and place children in the place of successor

delete 20, place 26 in 20, place 27 in 26

Successor does not have left subtree

Imbalance Factor : height of left subtree - height of right (4)



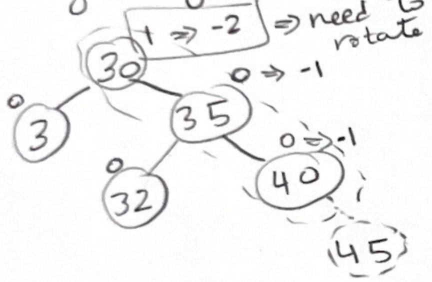
AVL Tree : Balanced BST where at any node the |Imbalance factor| < 2

↳ how to fix : Rotations

→ Start rotating after the first insertion causing balance factor ≥ 2
→ the first node to select case is where the |BF| ≥ 2

Cases

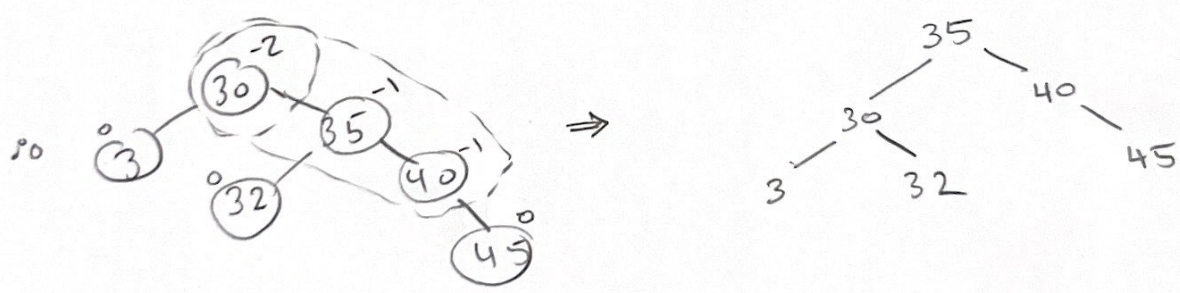
1) Right Right (RR)



⇒ take 30, and the following two children in the path of the imbalance.
i.e. 30, 35, 40

The two children are right right
∴ RR case.

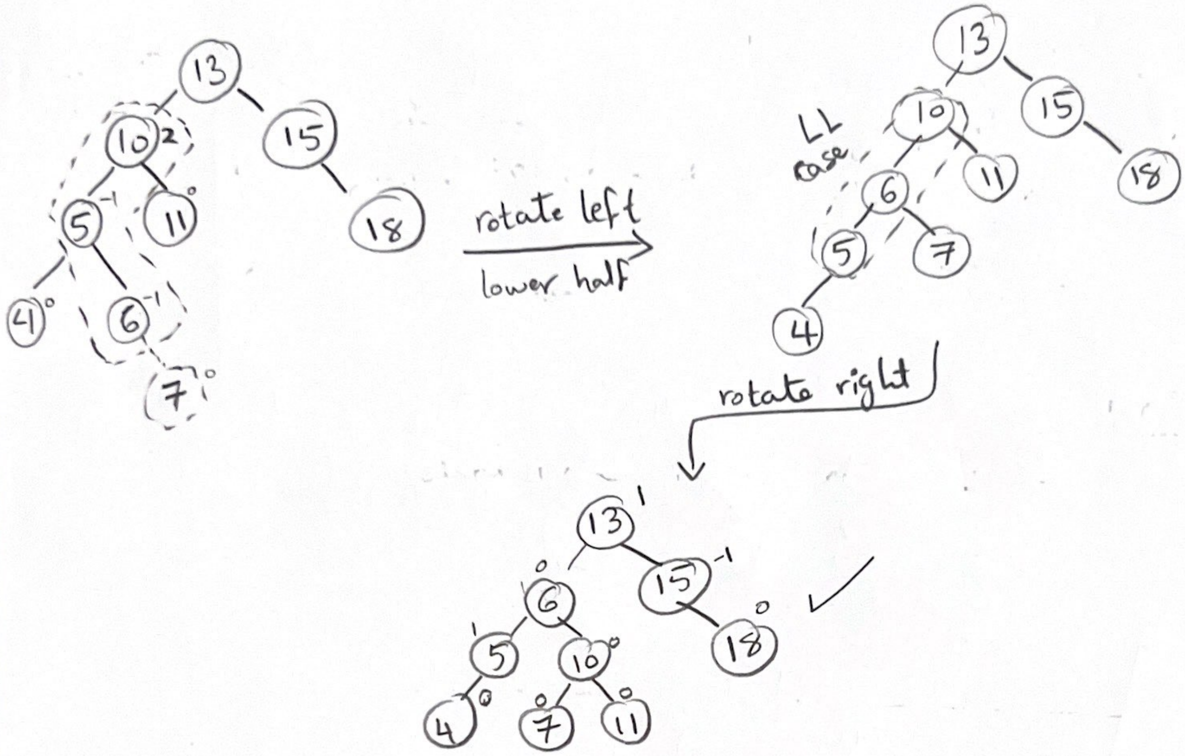
so



steps : 1- left rotate
2- left of 30 becomes at left
3- right of 35 becomes at right
4- left of 35 becomes at right of 30

2) Left Right

* Rotate left, then rotat right.



3) LL case :

opposite of what was done in RR

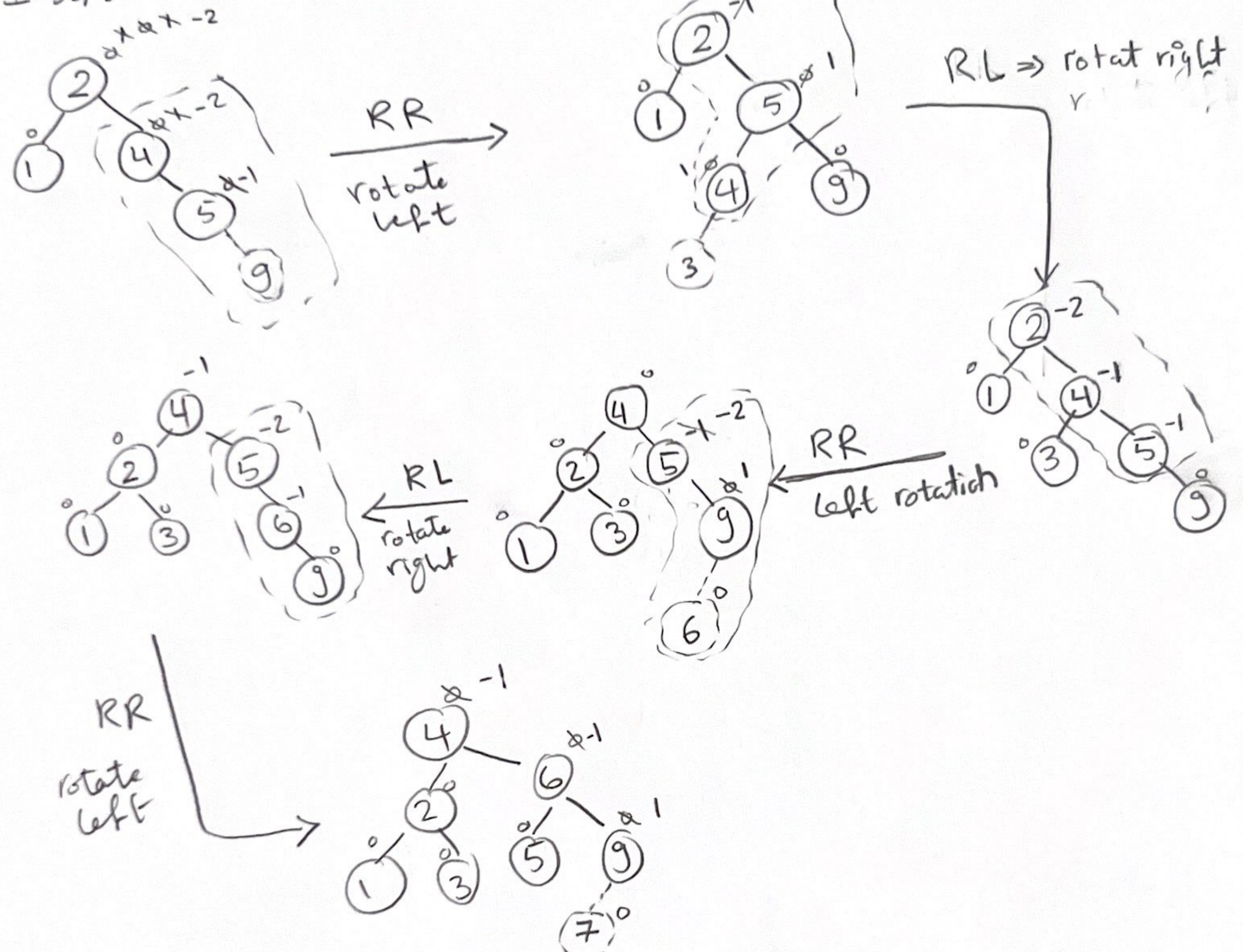
4) RL : opposite of LR

Delete

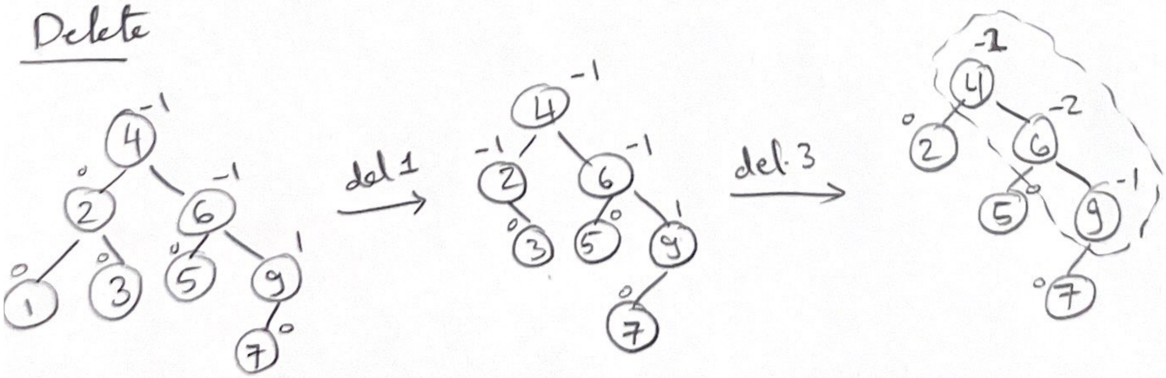
* delete as normal BST, then just check the balance factor.

ex: $\boxed{\text{AVL}}$

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



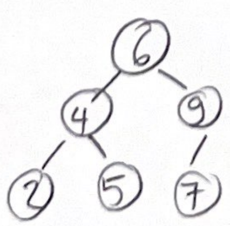
Delete



choose 4 and following two nodes in longest path. i.e. 6, 9

Balance

RR case



Red-Black Trees

BST with some properties : "aim to facilitate balancing"

1- Any node inserted acquire a red or black color

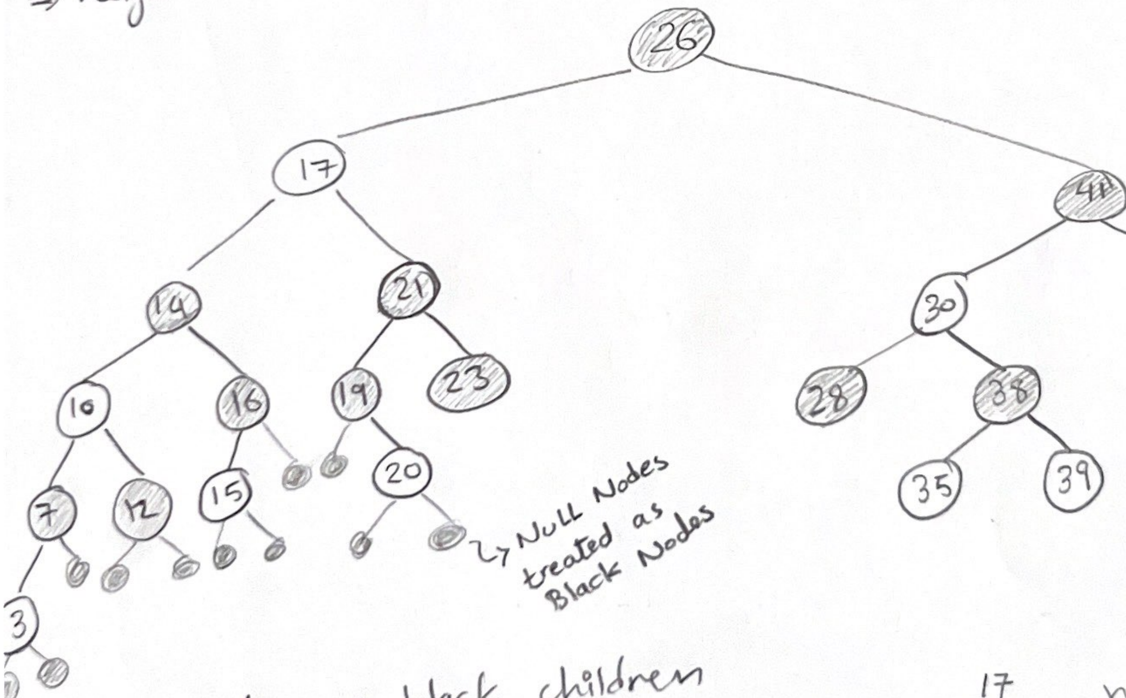
2- Root is always black

3- any nullptr is a black node.

insert 4- Any red node must have black children, otherwise we need a balancing step. "serves insertion process"

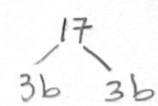
delete 5- For any node, every path from this node to any leaf have the same number of black nodes. "serves deletion process"

⇒ height of RB trees is at most $2 * \log(n+1)$



→ all red have black children

→ from any node → same # of black



nodes + null = 3 at any path

AVL vs. RBT

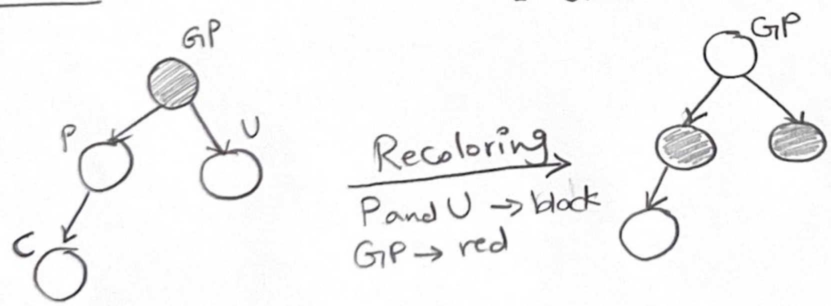
1- AVL is faster in search as less height (Balance Factor) ∴ more balanced
2- RBT is faster in insertion and deletion as it relaxes condition of balance, need only 1 bit red or black to save size. but in AVL, need 1 byte
RBT less balanced, ∴ less balancing operations. ∴ faster

Insertion:

→ Insert normally as BST tree
→ new node becomes red
→ children are black nullptrs
→ when does it break the properties? when parent is red.

Case 1 "Red Uncle"

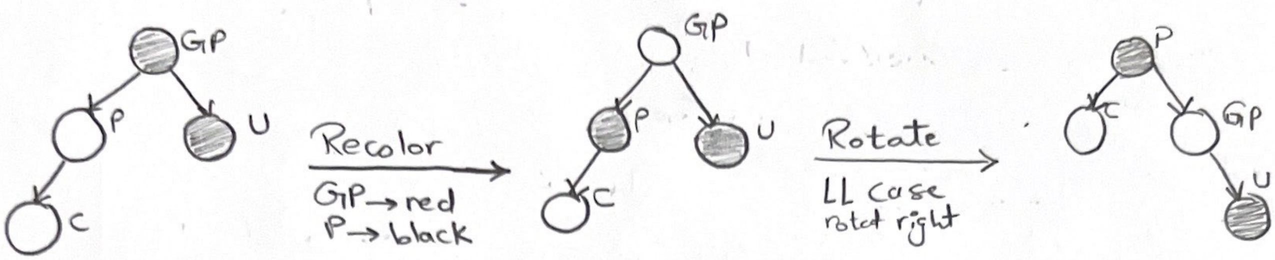
- New child is red
- parent is now red "property broken"
- Uncle is red



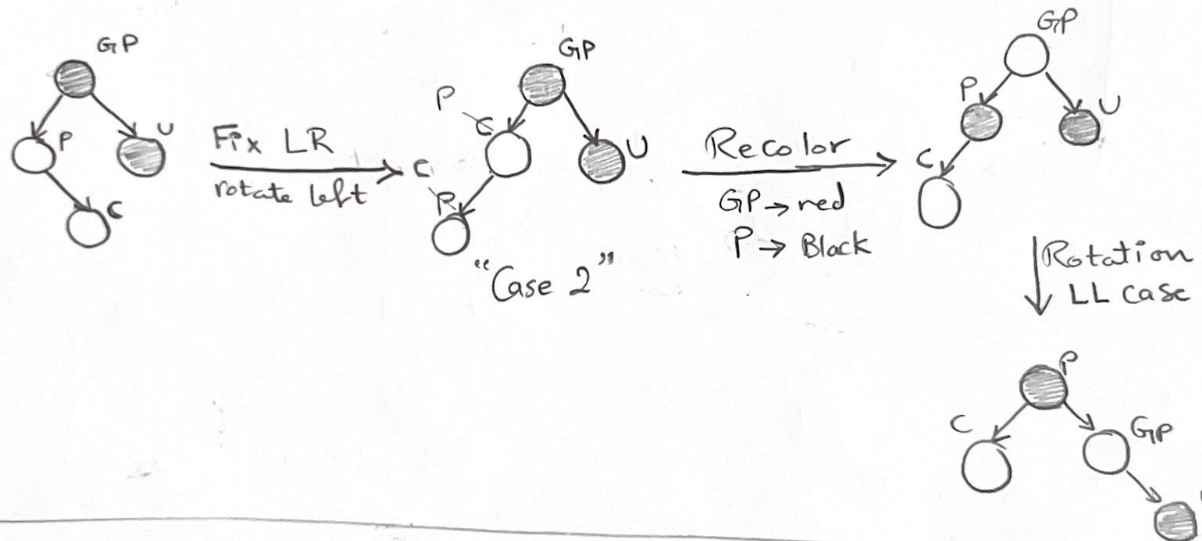
- if GP has red parent, then check again case according to uncle of GP
- if GP is root after all recoloring, recolor to black.

* Rotation in RBT is inserted red node + Parent + GP, rotation is like AVL

Case 2 Black Uncle



Case 3 "Black Uncle" Left Right



* Black uncle + Left Left or Right Right = Case 2 ∴ steps: 1) Recolor GP, P
2) Rotate

* Black Uncle + Left Right or Right left. = Case 3, ∴ steps: 1) Fix by rotation "converted to Case 2"
2) Recolor GP, P
3) Rotate

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