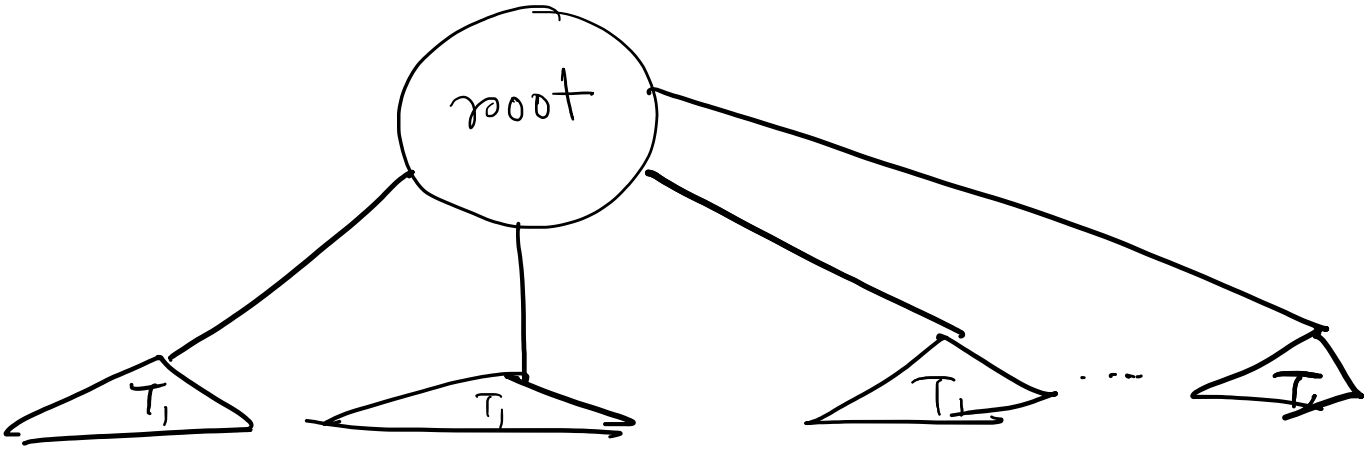
Chapter 4

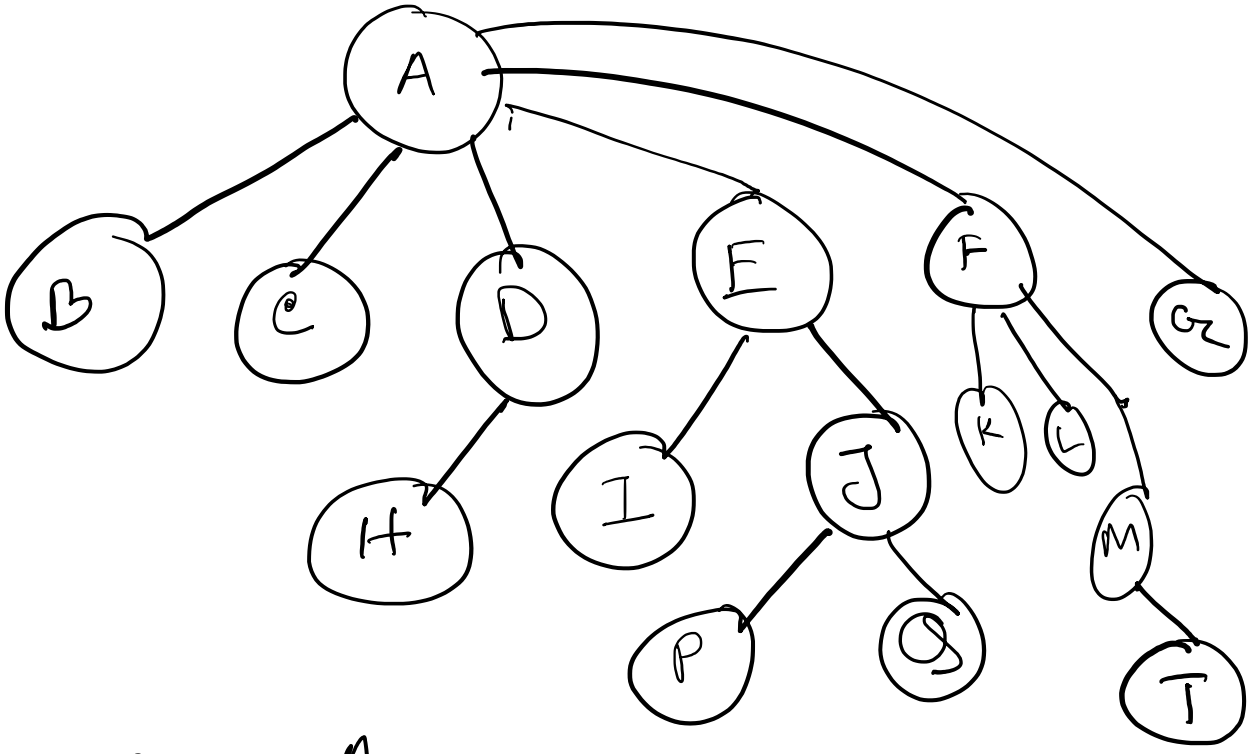
Tree



Collection of nodes and edges.

Recursively,

a root and its subtrees



Parent (D) = A

children (D) = {H}

children (E) = {I, J}

Ancestors (J) = {E, A}

Descendents (F) = {K, L, M, T}

Siblings (B) = {C, D, E, F, G}

Definitions

→ **Leaf node:** node having no children

→ **size:** total number of nodes in tree

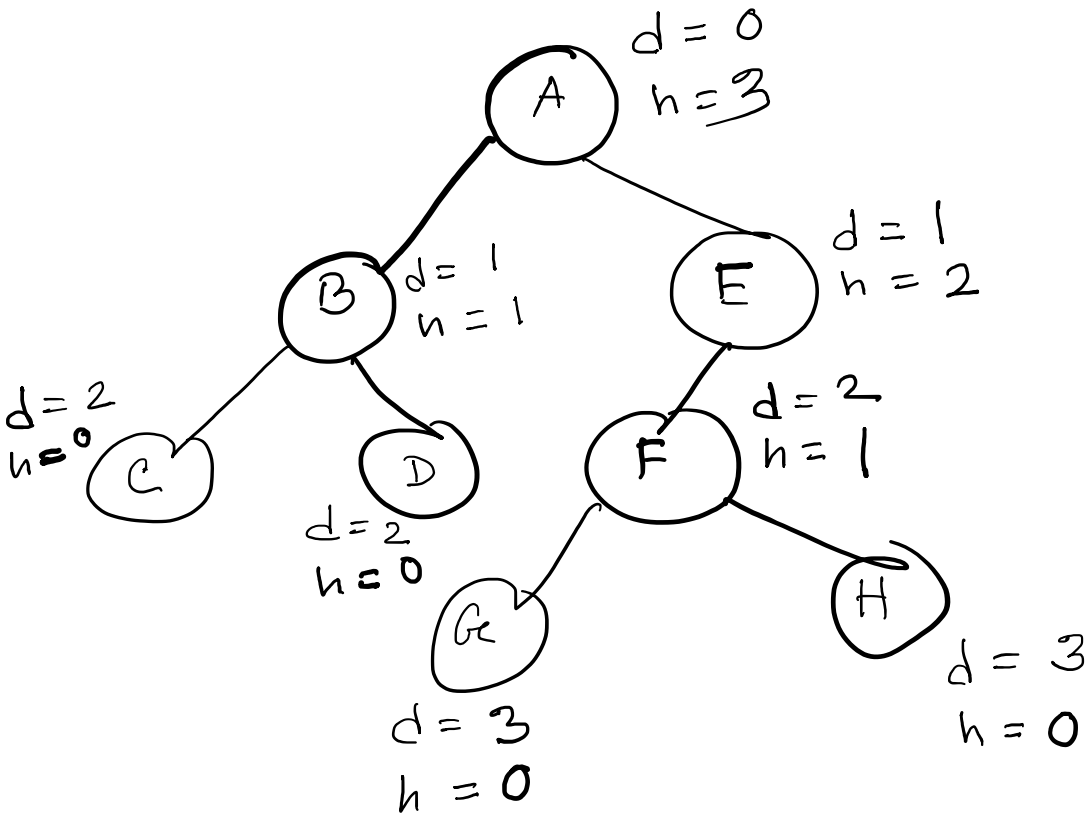
→ path$(s, d) = \{n_1, n_2, n_3 \ldots n_k : n_1 = s,$ $n_k = d$, parent$(n_{i+1}) = n_i$ for $1 \leq i \leq k\}$

→ Length of a path: is the number of edges along the path

→ Level: all nodes at a given depth share a level.

→ **Depth:** distance of node n from root in terms of the number of edges

→ **Height:** maximum distance of node n from its farthest leaf.



→ **Width:** number of nodes on the level containing the most nodes.

Binary Tree

min height = $\lfloor \log_2 n \rfloor$

max height = $n - 1$

n = total number of nodes

Array representation

parent (i) = $(i-1)/2$

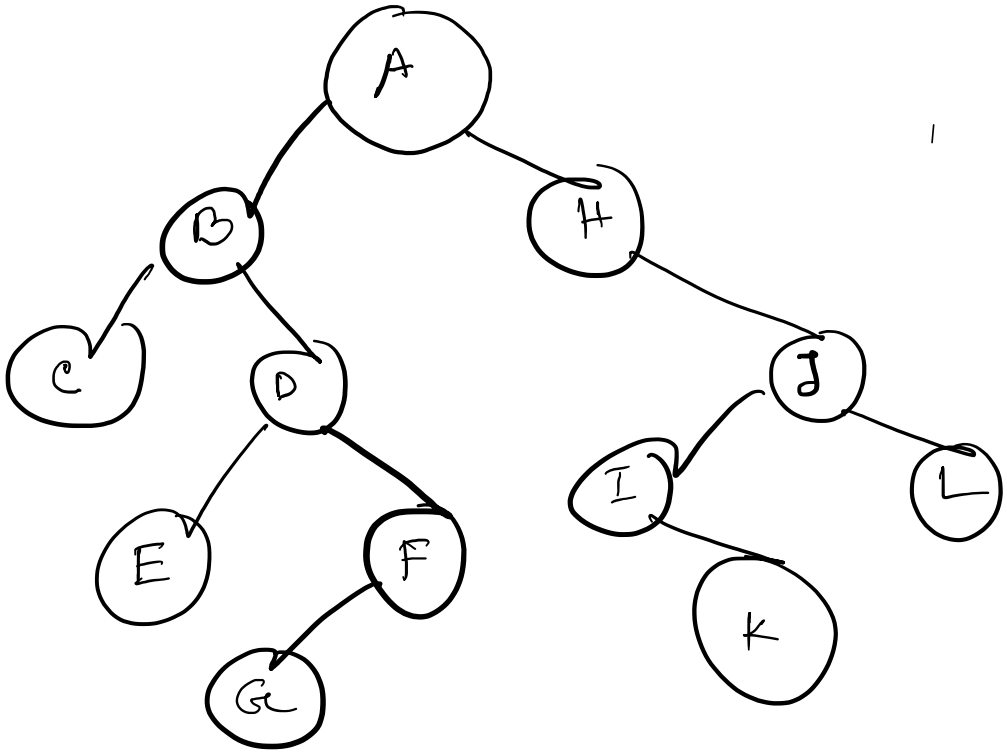
Left (i) = $2i + 1$

righ (i) = $2i + 2$

Tree traversal:

→ Breadth-first: visit nodes level by level, from left to right

→ Depth-first:
- → pre-order: root, left, right
- → in-order: Left, root, right
- → post-order: left, right, root



Breadth-first: A B H C D J E F I L G K

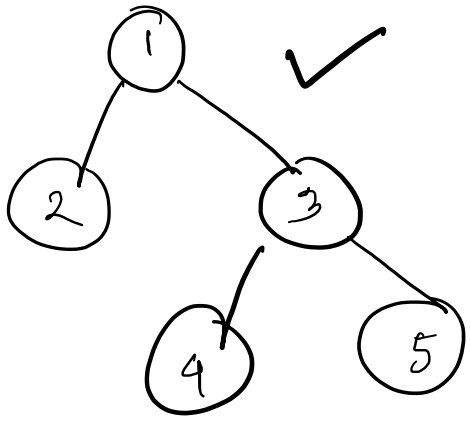
Pre-order: A B C D E F G H J I K L

In-order: C B E D G F A H I K J L

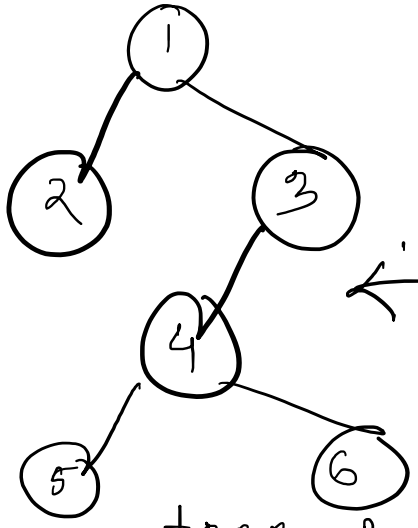
Post-order: C E G F D B K I L J H A

Full Binary Tree:

Each interior node (non-leaf) node has two children



tree A

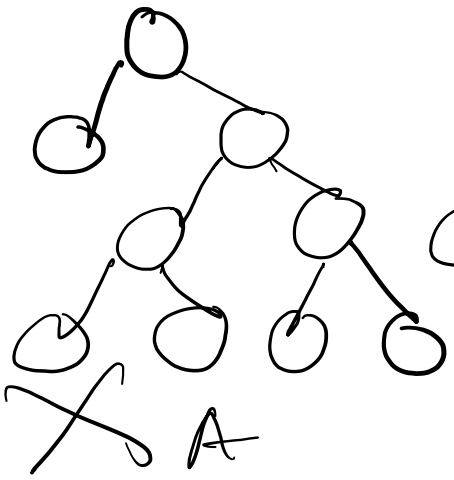


← not a Full B. Tree

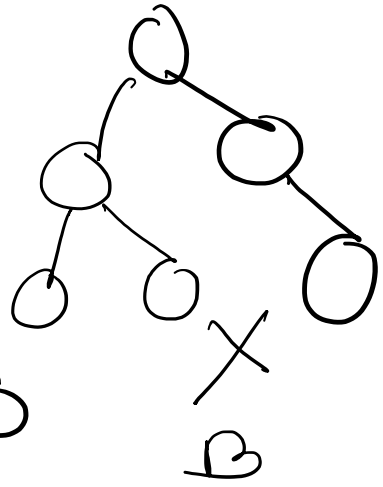
tree B

Perfect Binary Tree

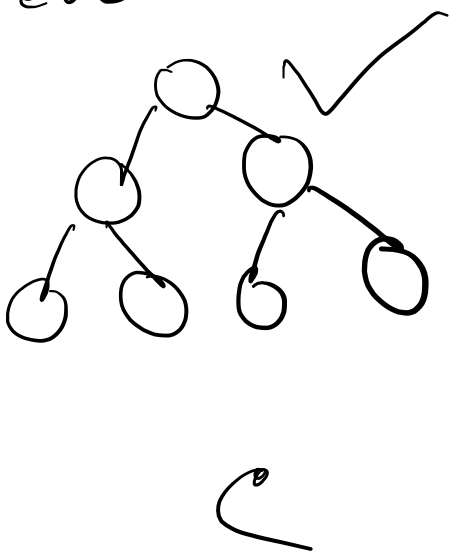
A full binary tree with all leaves at the same level.



A



B



C

Binary Search Tree

Operations

1. contains
   - ↳ boolean
2. findMin / findMax
   - ↳ for min keep going to the left
   - ↳ for max keep going to the right
3. insert
   - ↳ if x is not found in the tree (contain returns false), insert in appropriate path
4. delete

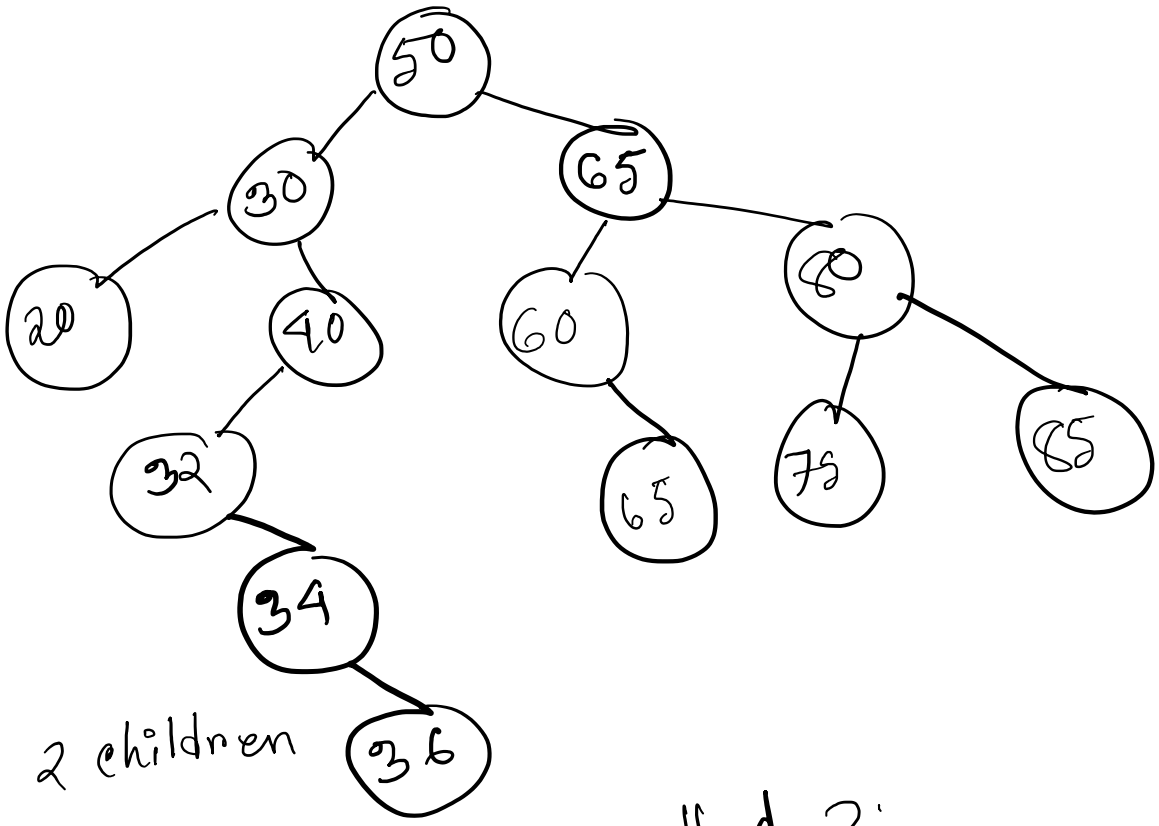
   3 cases

case 1: Leaf node

→ delete the node

case 2: 1 child

↳ delete the node and connect with the parent of the deleted node.



Case 3: 2 children

method 1:

a) Find min of the right subtree

b) copy the value in targetted node

c) remove duplicate

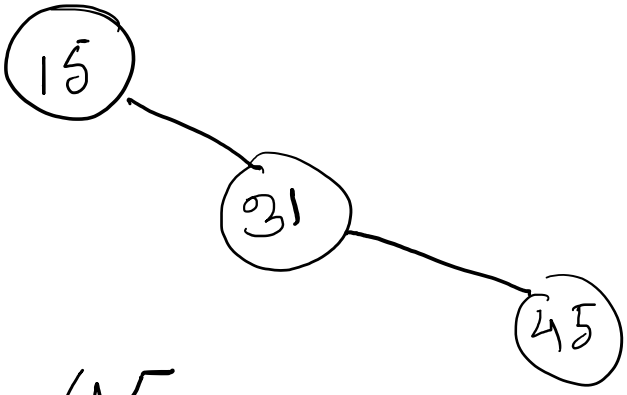
method 2:

a) Find max of the left subtree

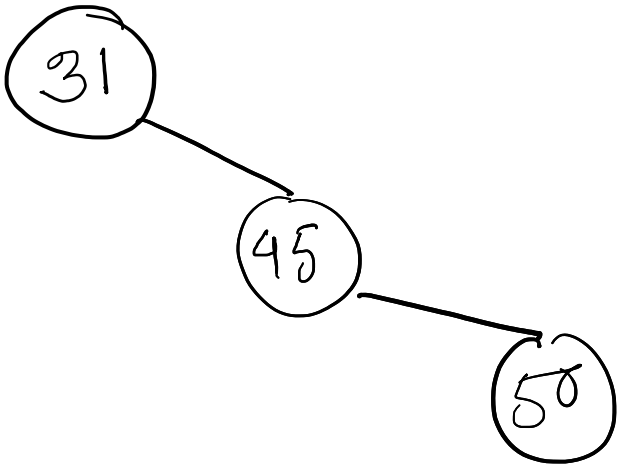
b) and c) same as method 1.

Balancing

15, 31, 45



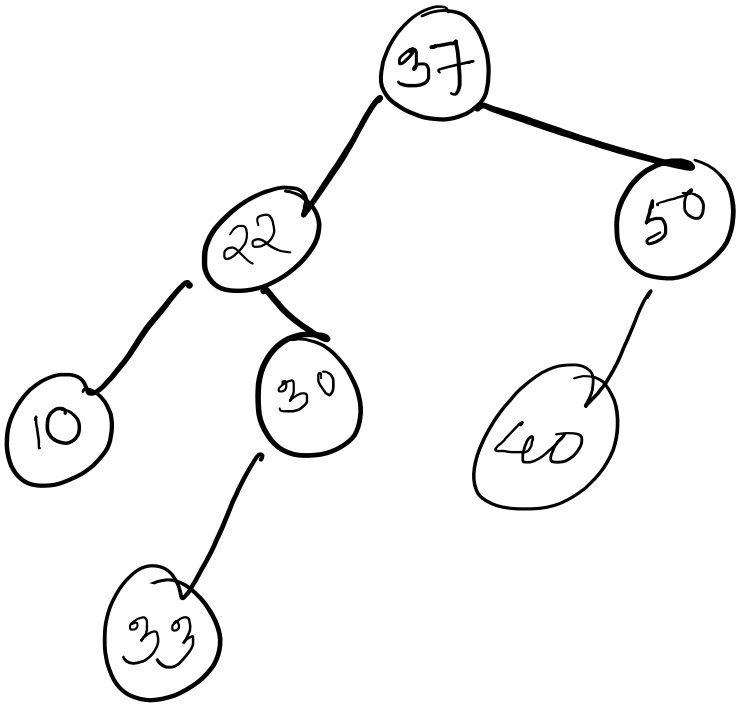
31, 15, 45



AVL Trees

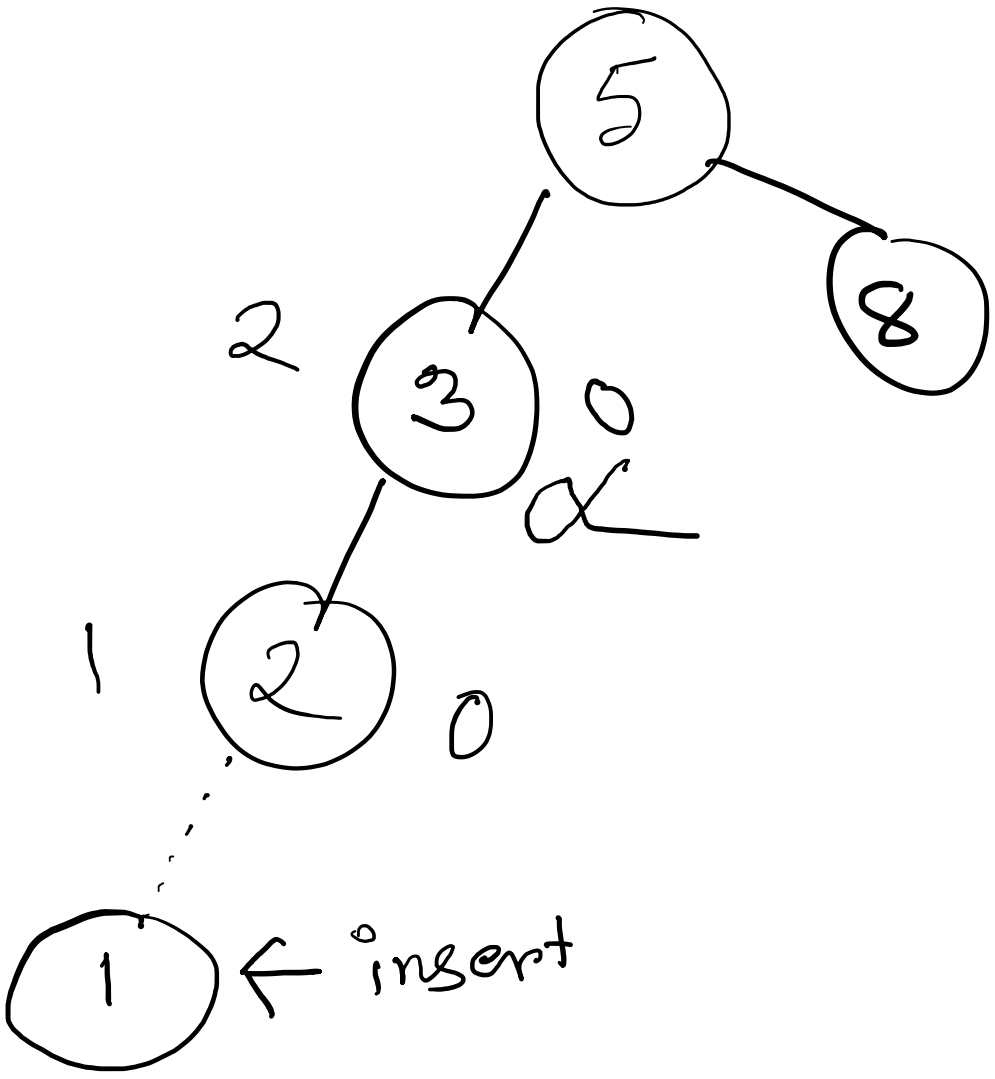
A binary search tree where for every node the height of the left and right subtrees can differ by at most 1.

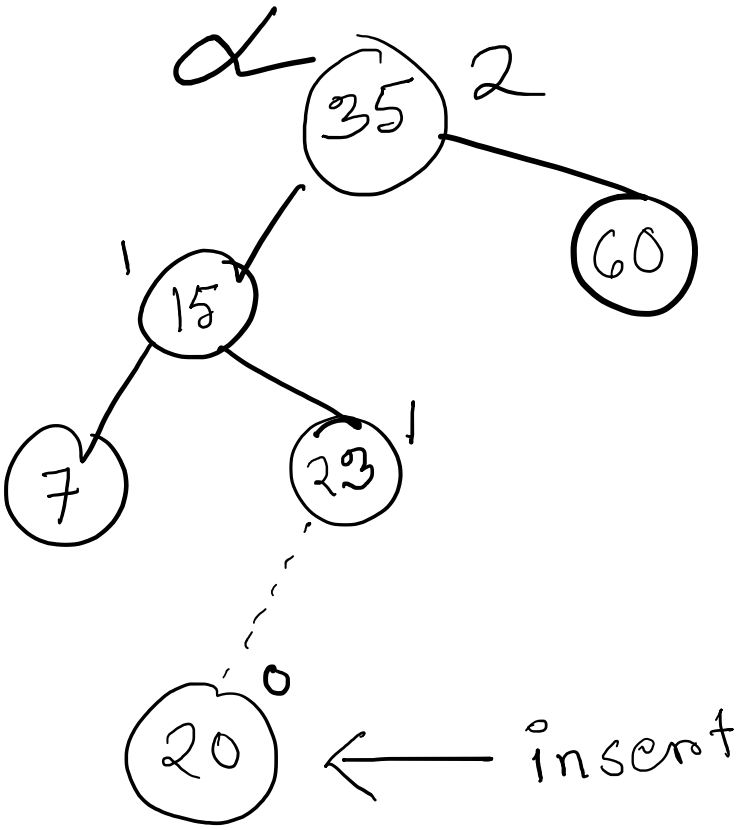
AVL = Adelson-Velskii and Landis



Four cases of unbalanced insertions

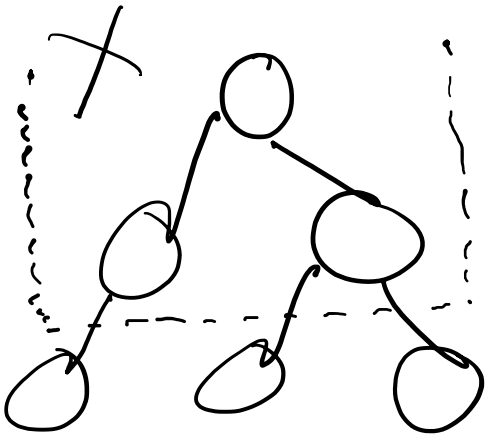
Let α be the node that become unblanced



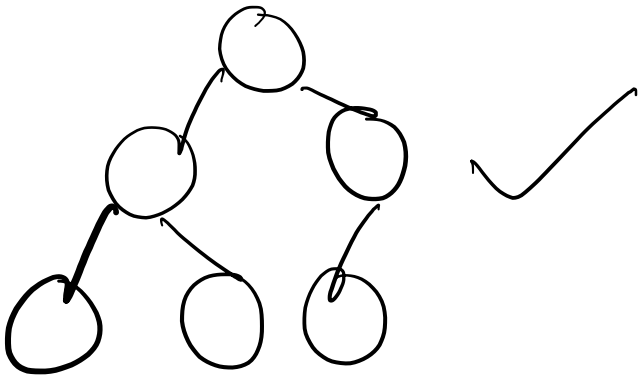

α
35
2
60
1
15
7
23
1
0
20
insert

Complete Binary Tree

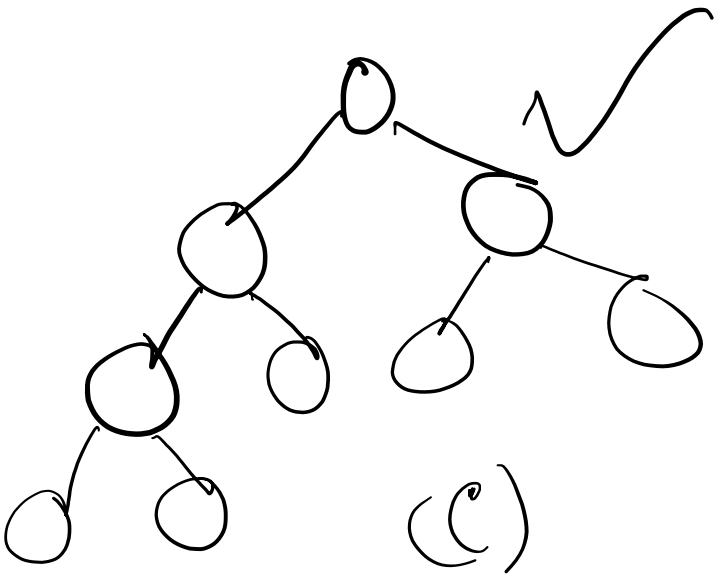
A binary tree of height h, where the tree is a perfect binary tree down to height h-1 and the nodes at the lowest level are filled from left to right



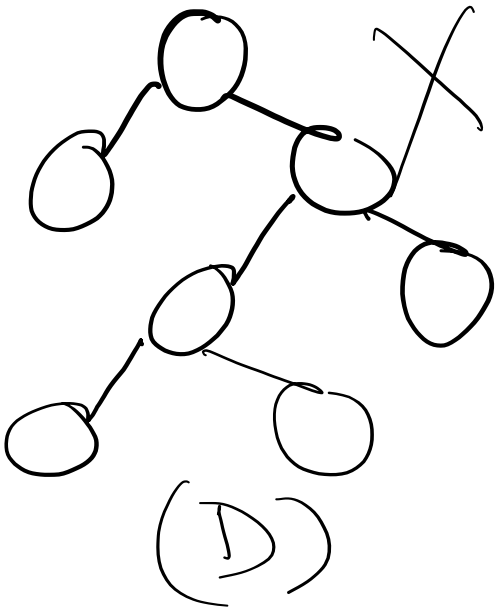
(A)



(B)

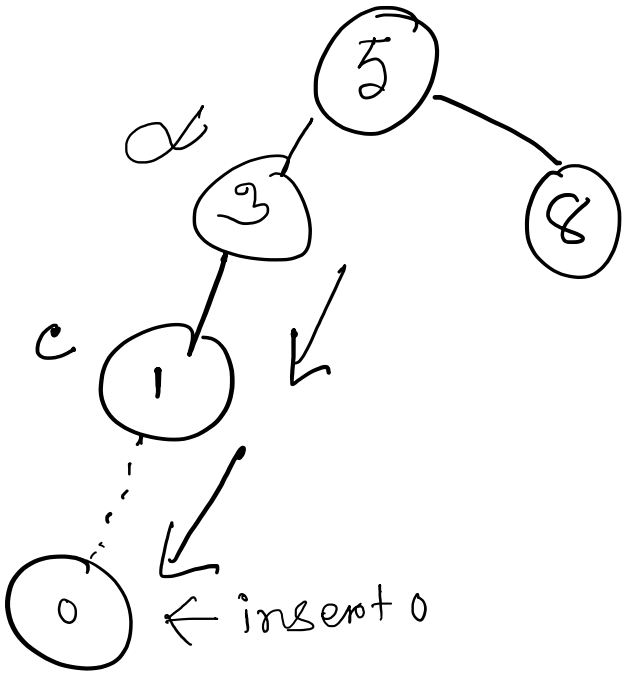


(C)



(D)

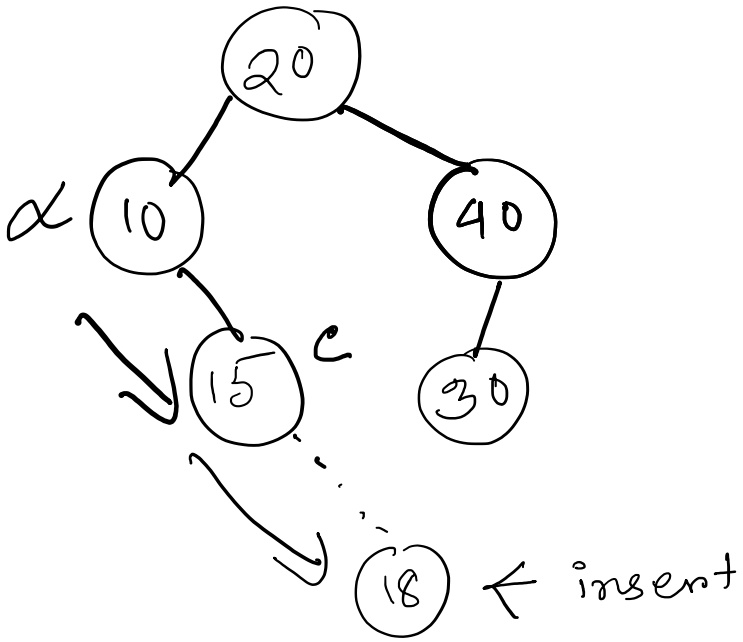
Case 1: insertion into the left subtree of the left child (c) of α



left - left

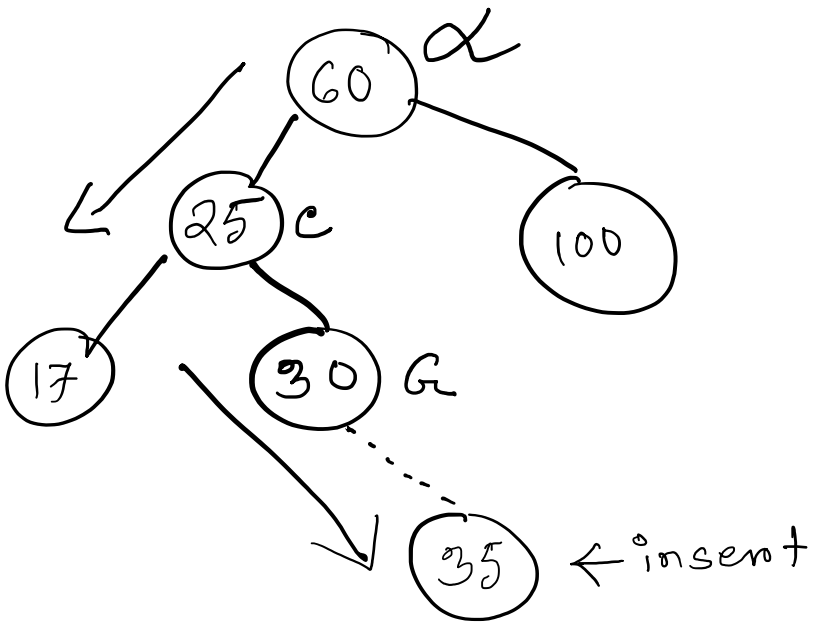
zig. zig

Case 4: Insertion into the right subtree of the right child of α



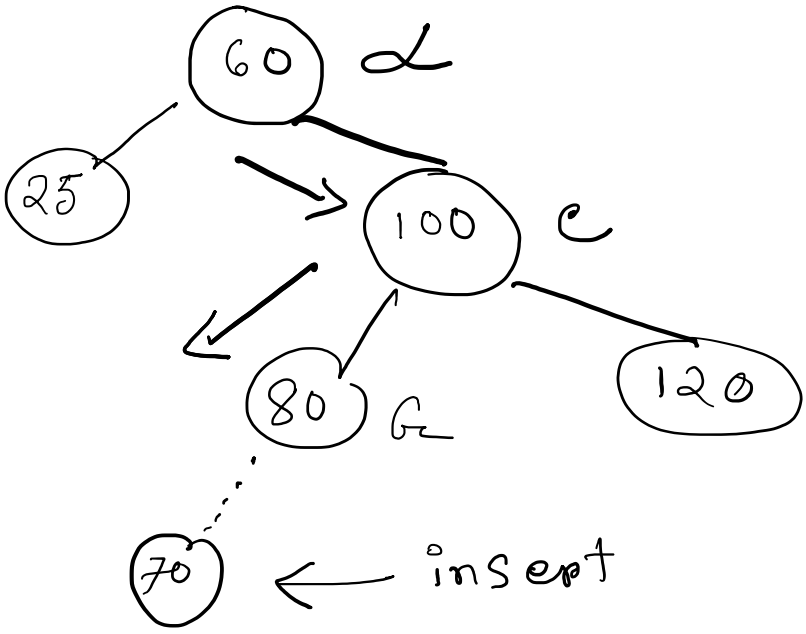
right - right

Case 2: insertion into the right subtree (G) of the left child (c) of α.



zig-zag
left-right

Case-3: insertion into the left subtree of the right child (c) of α.

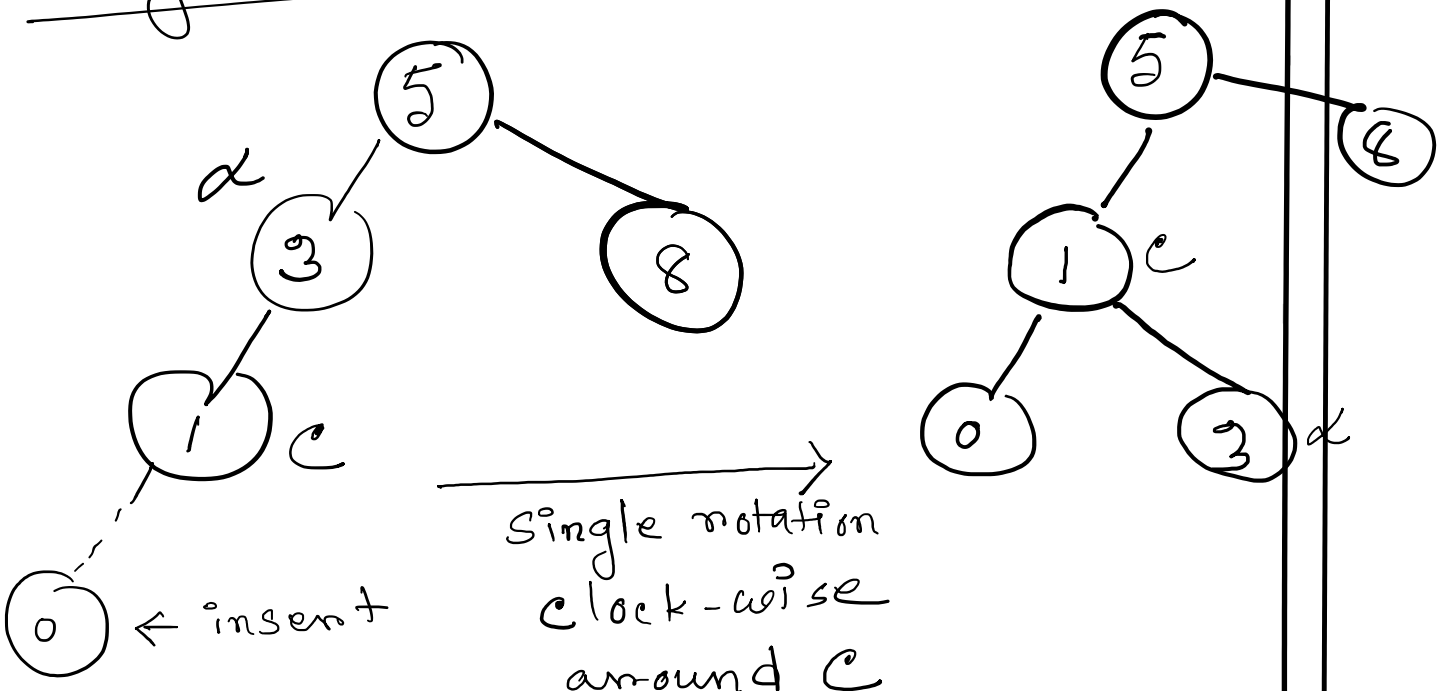


right-left

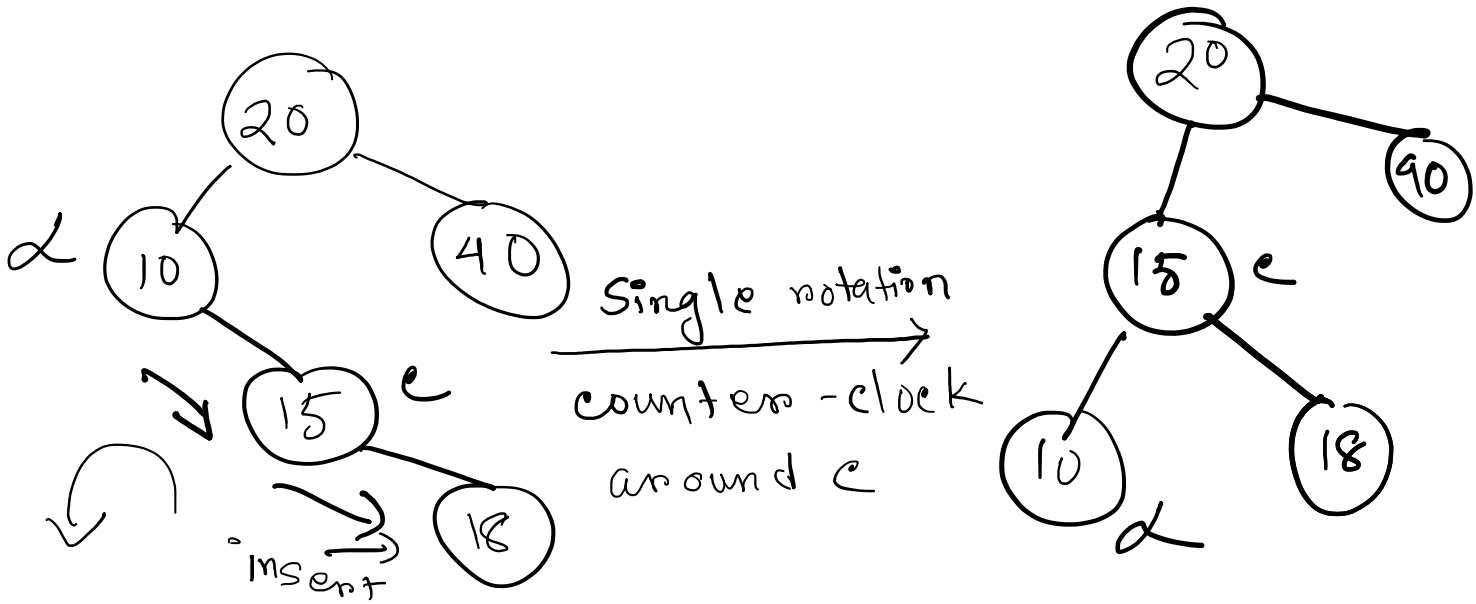
Balancing Strategies:

1. Single Rotation: For case 1 and 4 (left-left or right-right)
2. Double-Rotation: For case 2 and 3 (left-right or right-left)

Single Rotation:

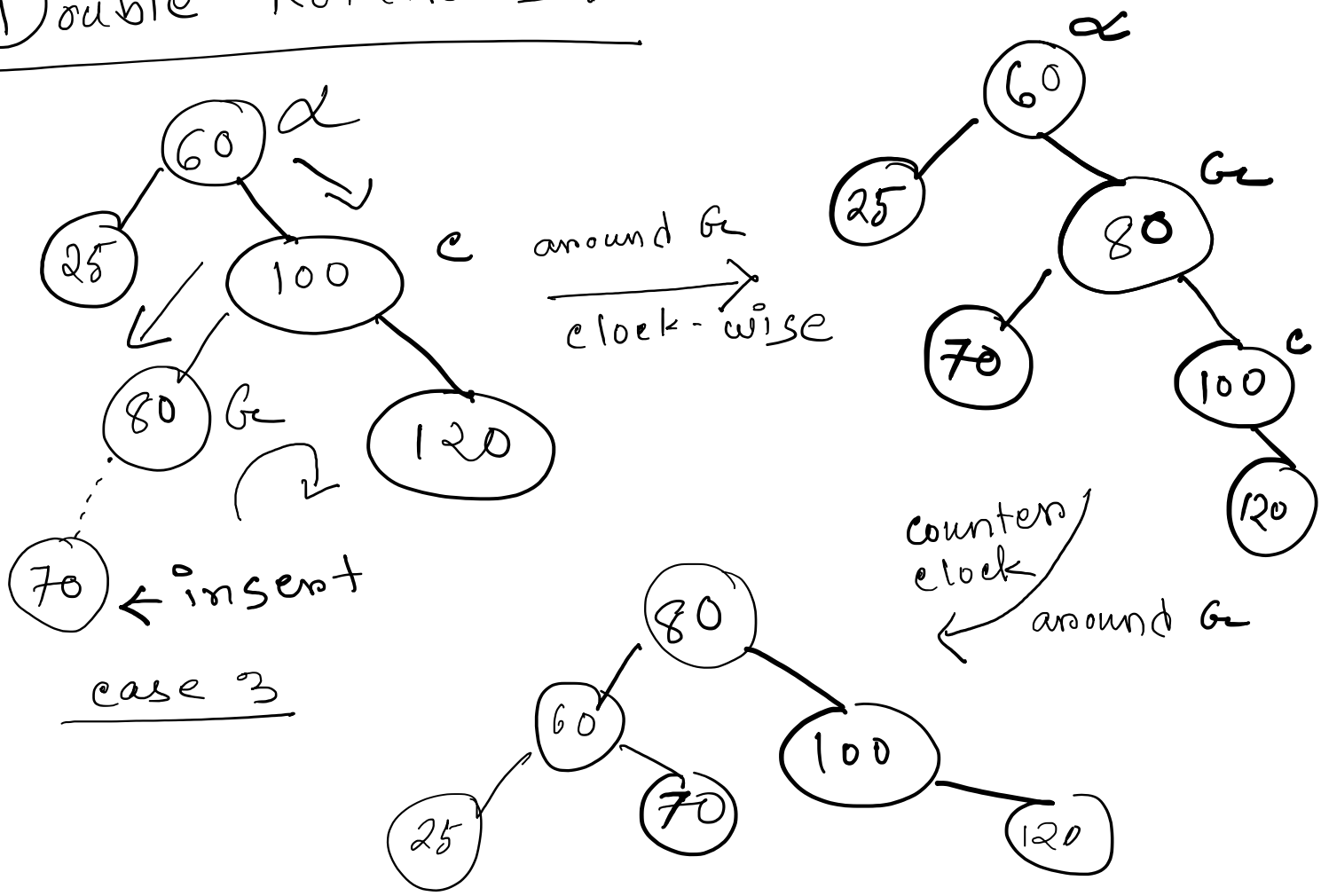


Case 1

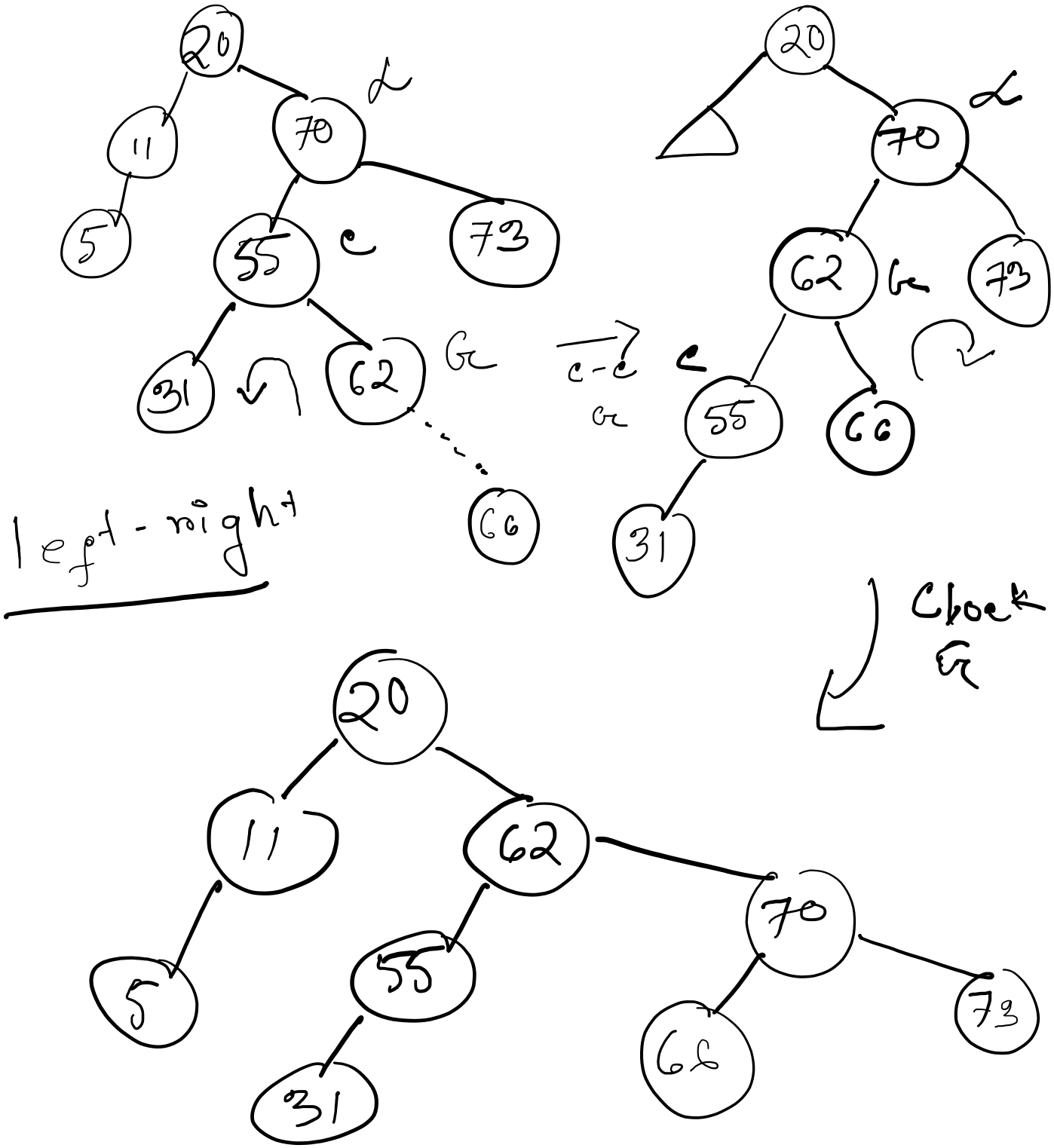


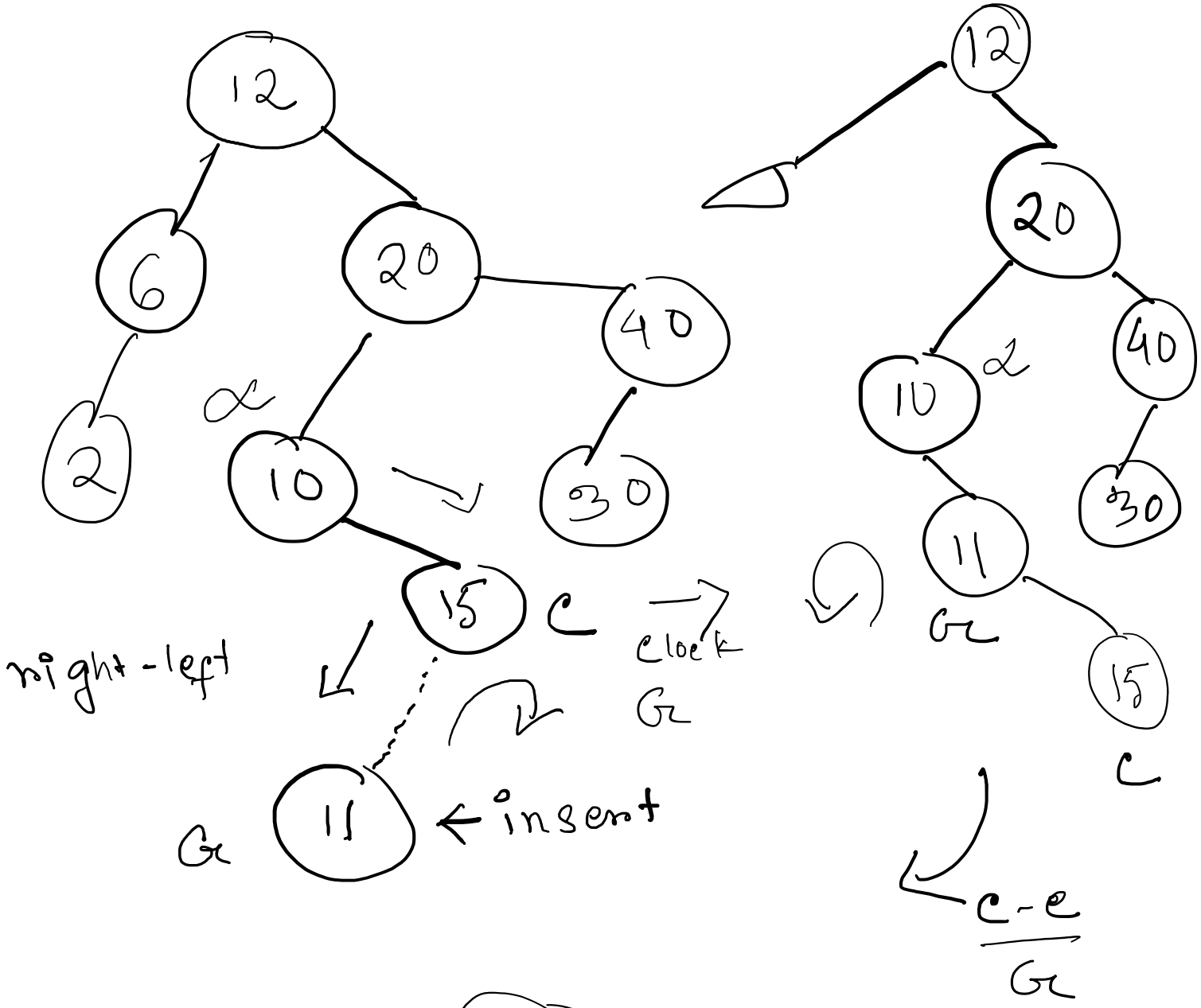
Case 4

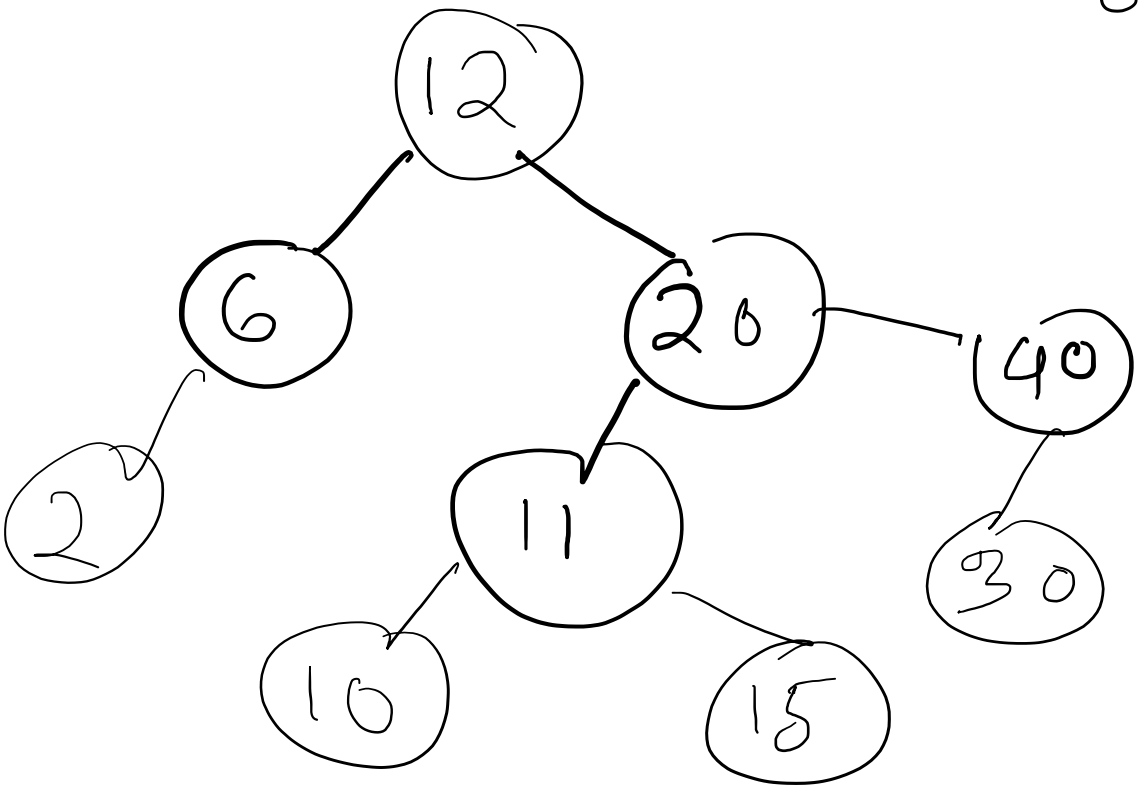
Double Rotations:



case 3


20
11
70
5
55
73
31
62
66
d
c
Gr
left - right
c-c
or
20
70
62
73
55
66
31
d
Gr
Clock
Gr
20
11
62
5
55
70
31
66
73


12
6
20
2
10
40
30
15
11
right-left
clock
insert
12
20
10
40
11
30
15
c-c

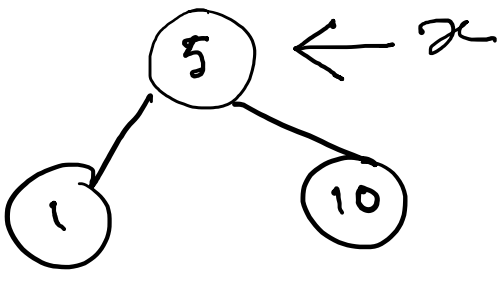

12
6
20
2
11
40
10
15
30

Splaying

The basic idea of Splay tree is that after a node is accessed, it is pushed to the root by a series of AVL rotations.

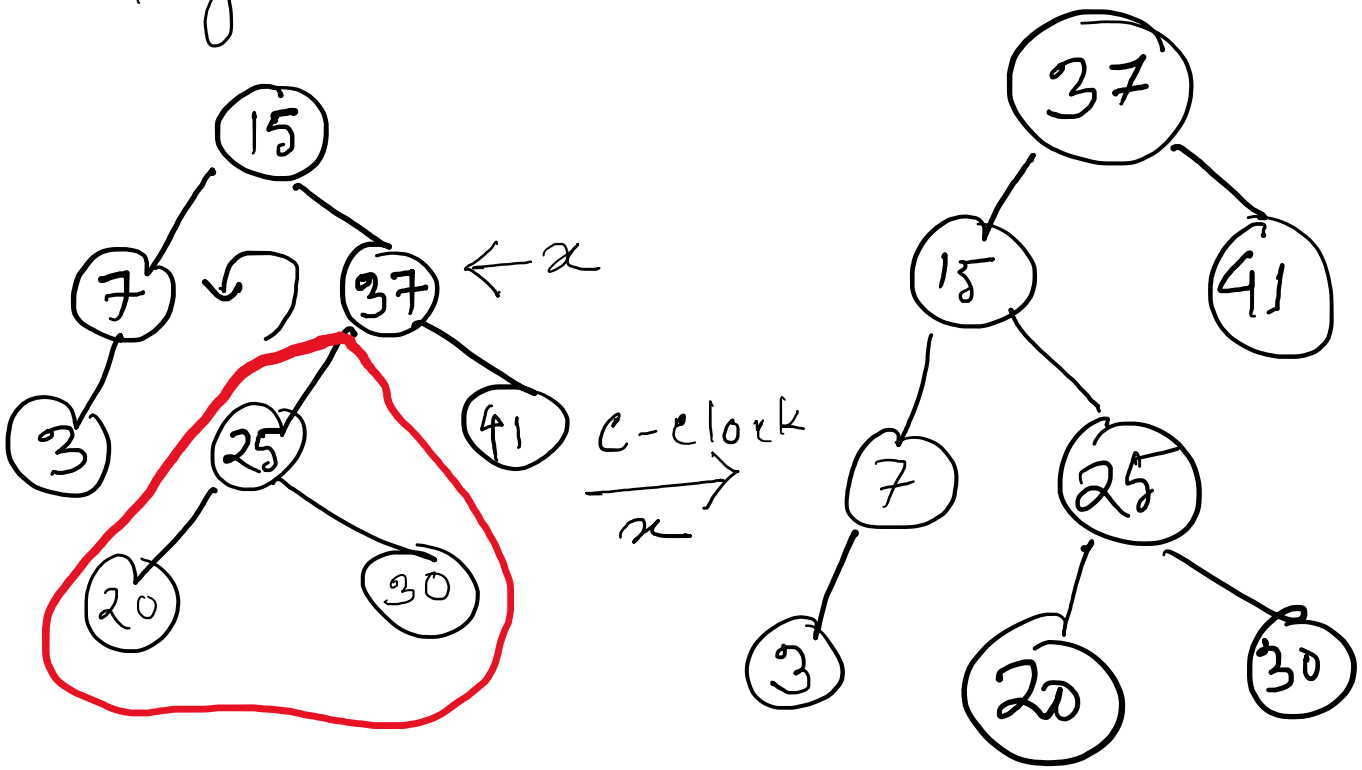
Let the last accessed node be x.

<u>case 1</u>: x is the root of the tree, do nothing.



Case 2: Parent of x is the root of the tree.

Simply rotate x and the root

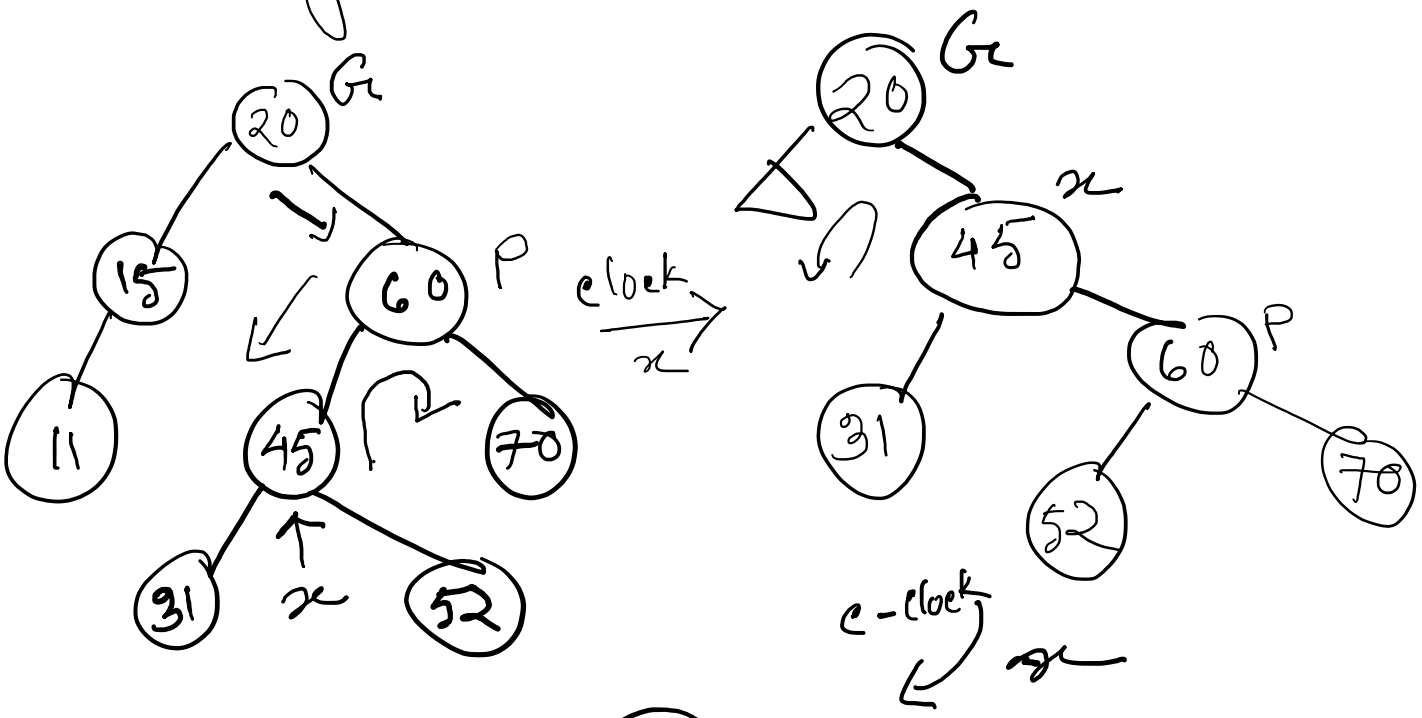


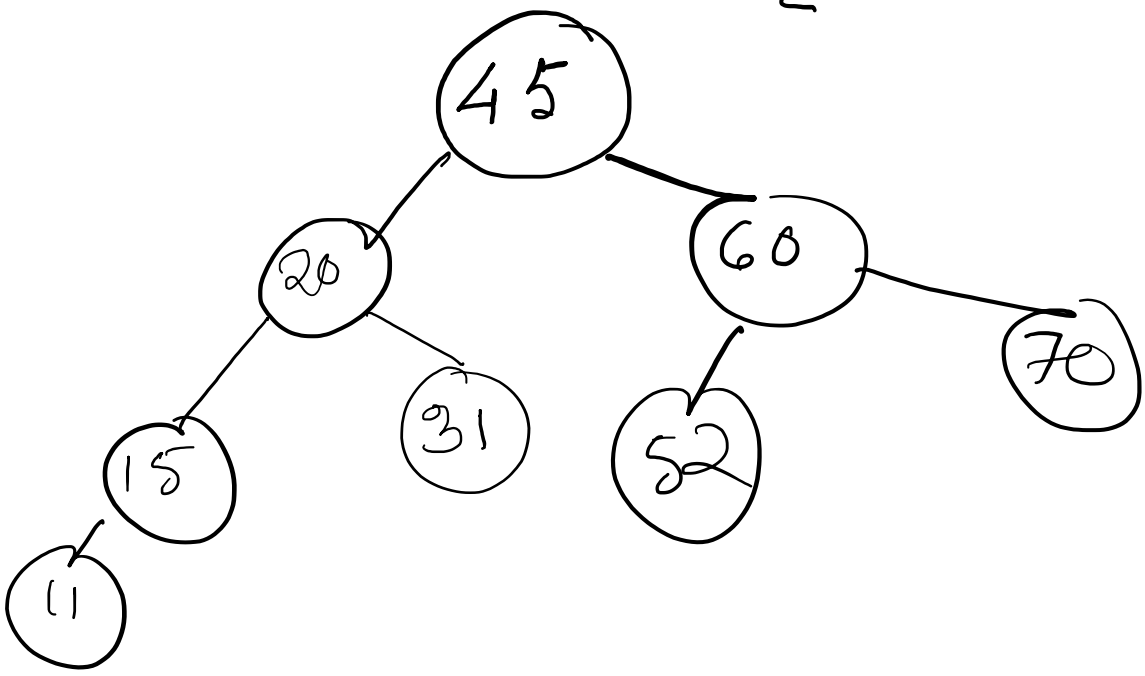
Case 3: Parent of x, denoted as P is not the root of the tree. So, x also have a grand-parent.

Two cases —

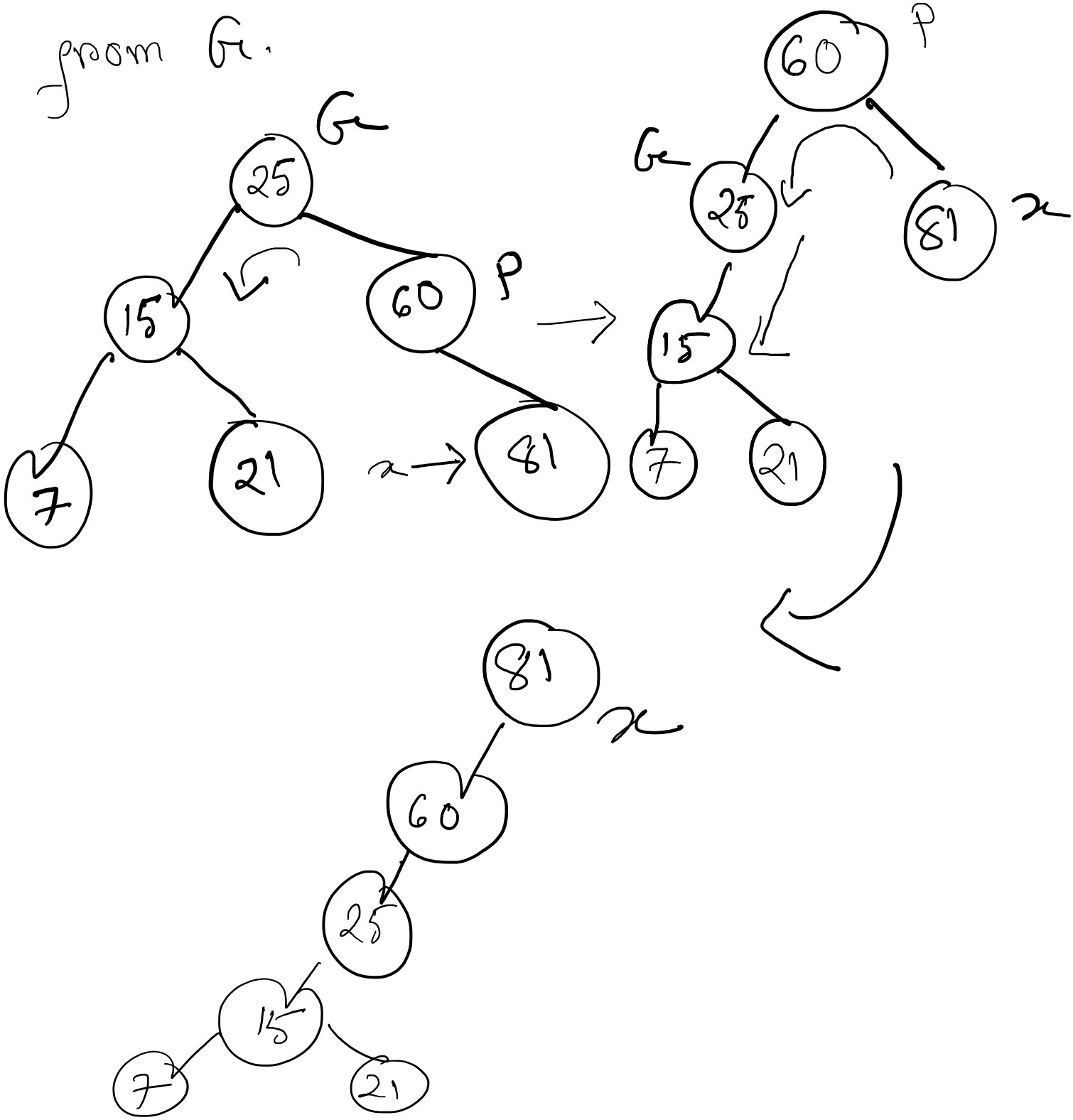
(a) Zig-Zag: x is right child and P is the left child (or vice-versa)

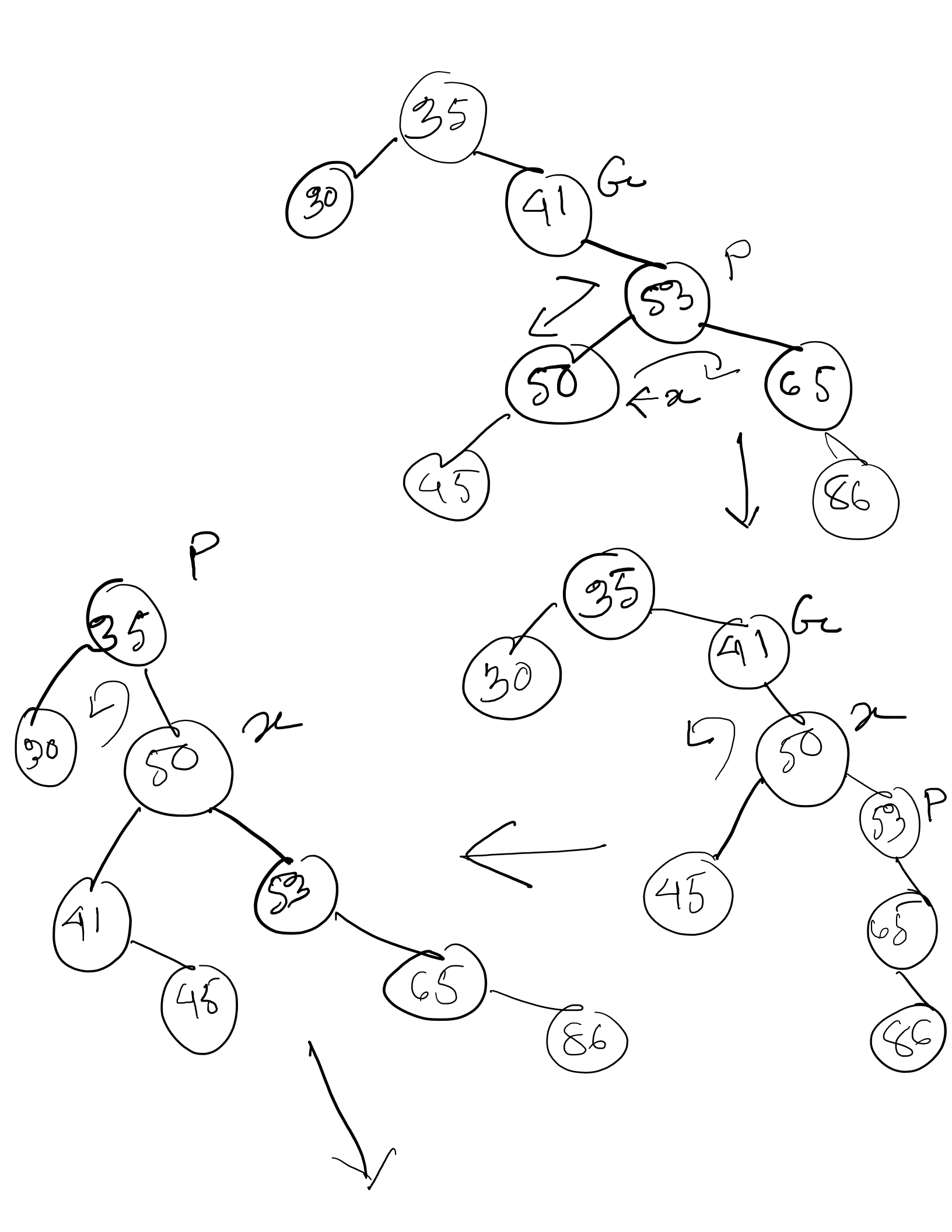
Perform a double rotation exactly like AVL tree. (around x)

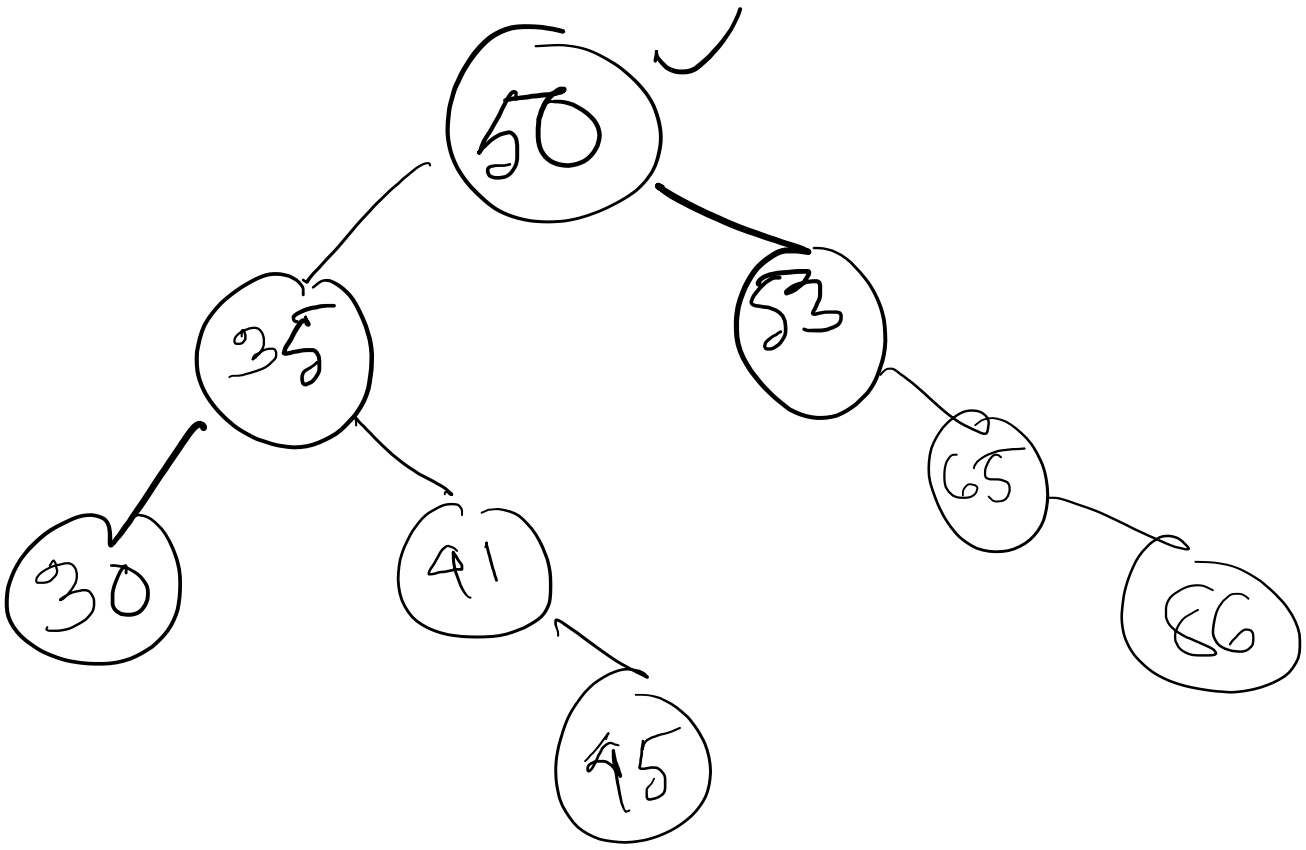




(b) Symmetric: x and P both are left children (or, right). Rotate twice downwards starting from G.




35
30
41
G
53
P
50
x
65
45
66
P
35
30
50
x
41
53
45
65
86
35
30
41
G
50
x
53
P
45
65
66



Red-Black tree:

A historical alternative of AVL tree for balanced binary search tree.

Properties:

→ Every node is colored either red or black.

→ The root is black

→ Null references (sentinel / leaf) nodes are black

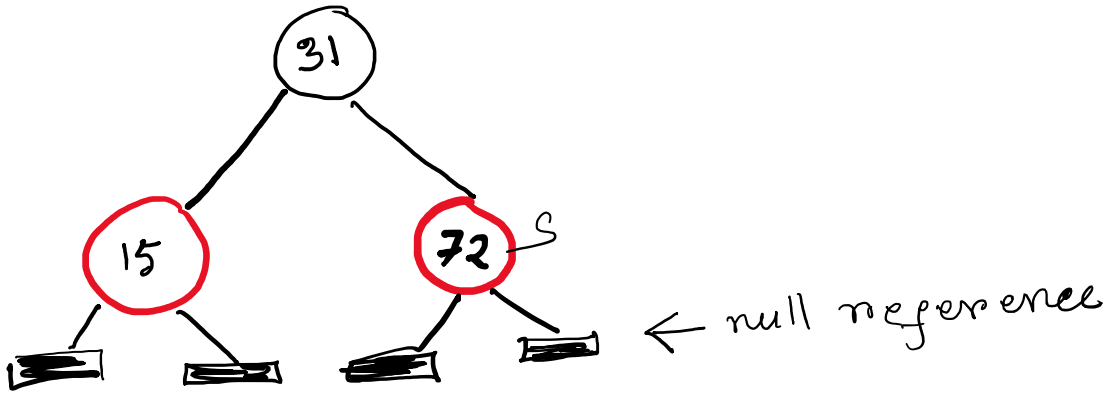
→ if a node is red, its children must be black

→ Every path from a node to a null (leaf) reference must contain same number of black nodes (black height).

→ Coloring rules ensure that a Red-black tree's height is at most $2 \log (N+1)$.

→ Operations take $O(\log_2 N)$ time in worst case

→ Newly added node is red.



Red - black
Black - black
Black - Red
Red - Red ✗

Let, the newly added node be x

Parent of x is P

Sibling of P is S

Grandparent of x is G

Case 1: If color (P) == black, do noting

Case 2: if color (P) == Red, thus, color (x) must be black. Color (S) can be black or red.

Case 2(a): color (S) == black

Case $2a_1$ (Symmetric): x and P both are left (or right) children.

→ perform single rotation around P

→ change color of the root of subtree to black

→ change color (G) to red.

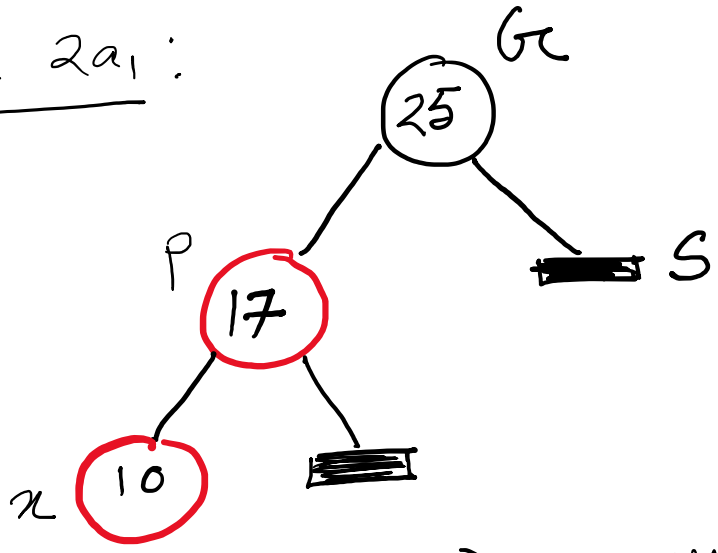
Case $2a_2$ (Zig-zag): x is the left child and P is the right child (or, vice-versa)

→ Perform double-rotation around x

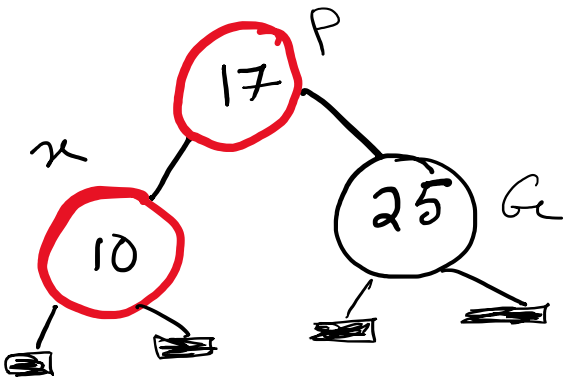
→ Color root of the subtree to black

→ color (G) to red.

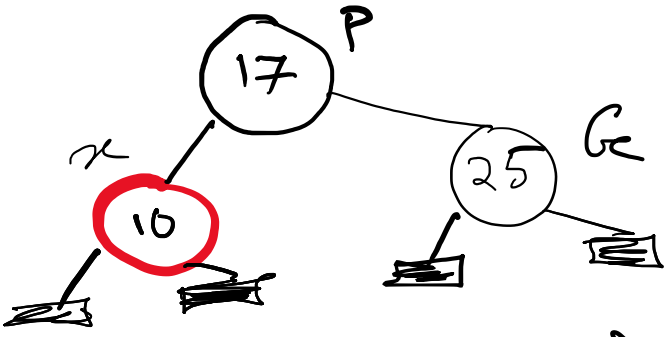
Case 2a1:



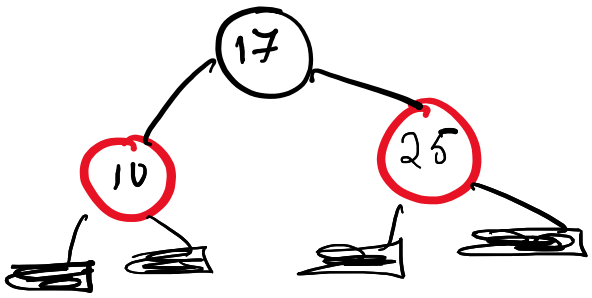
→ clock-wise rotation around P



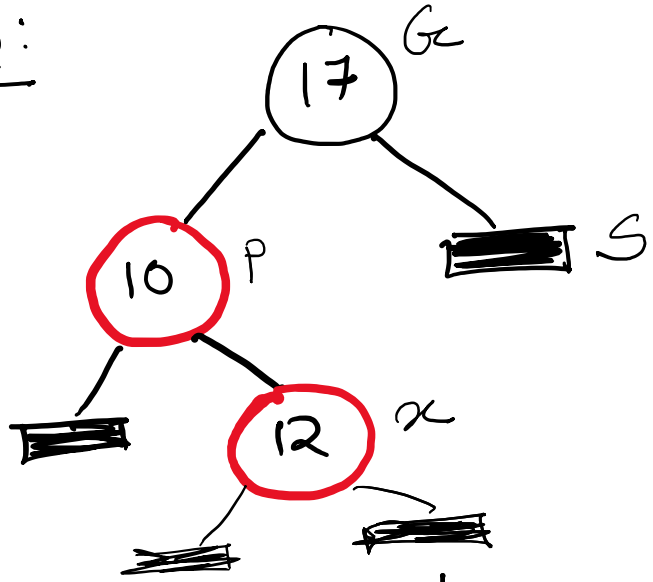
→ change the root of the subtree to black (17)



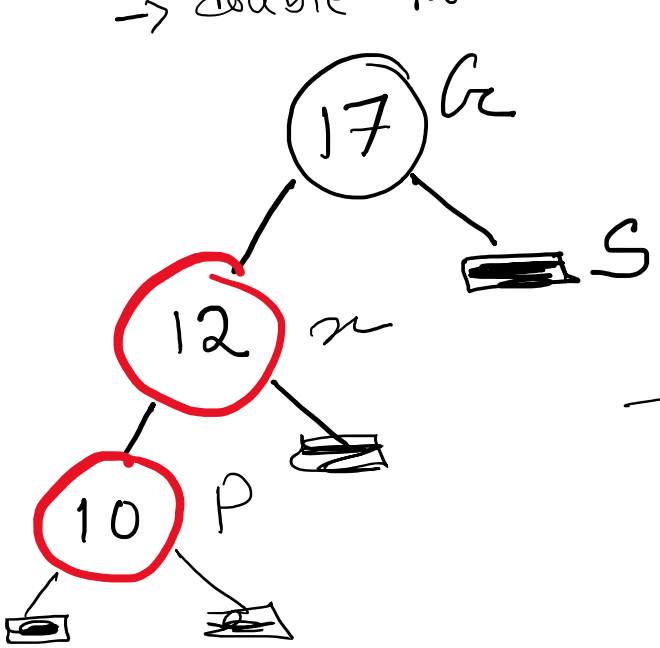
→ change the color of Gr(25) to red

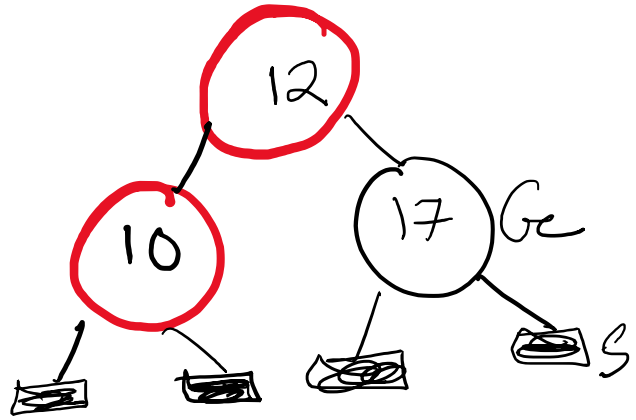


Case 2a2:

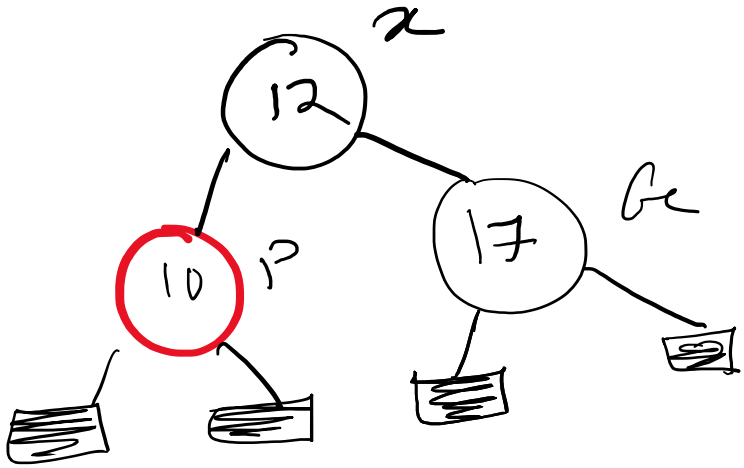


→ double rotation around x

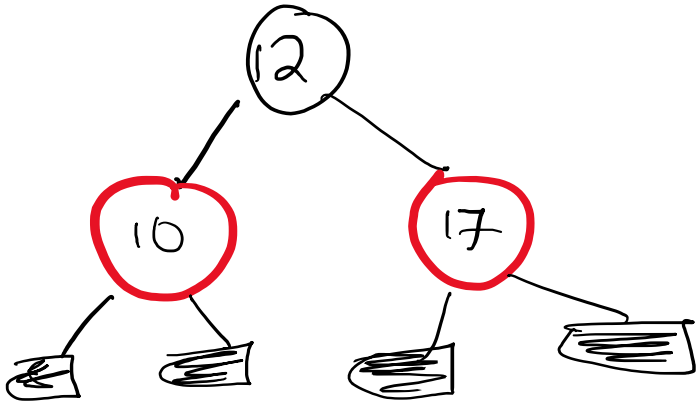




→ color the root(12) of subtree to black



→ color Ge (17) to red

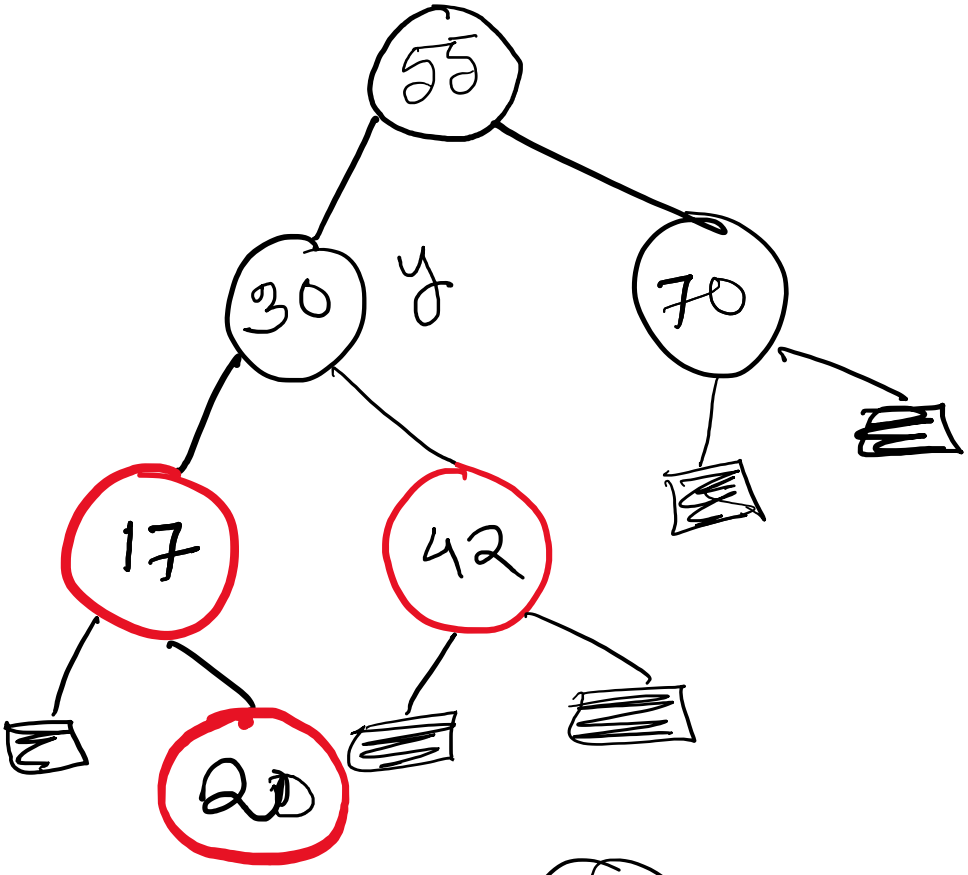


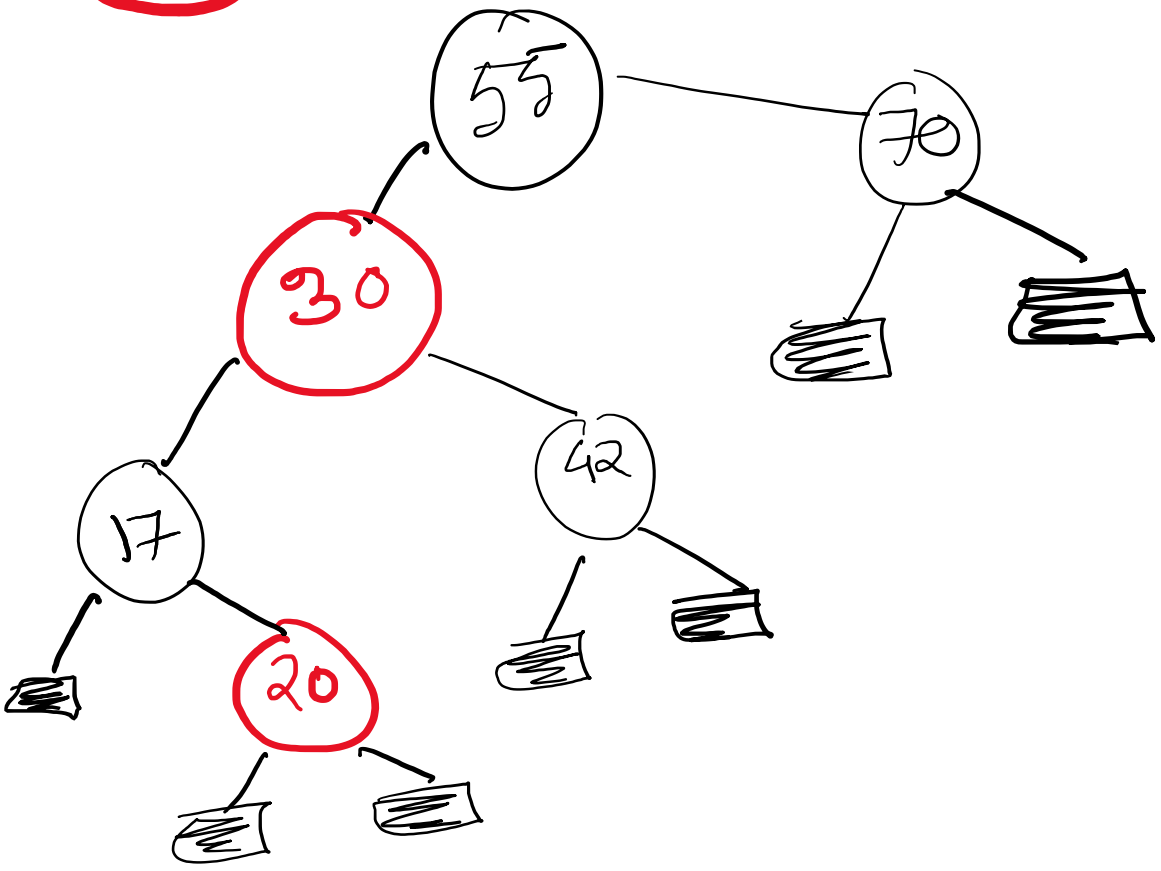
Case 2(b): color (S) == Red

→ Top-down recoloring

→ On the way down, when we see a node y that has two red children, we make y red and the two children black.

→


55
30
y
70
17
42
20


55
70
30
42
17
20