AVL-trees

Motivation:

In Binary search tree (BST) data structure, the operations Insertion, deletion, Search takes $O(h)$ time, where h is the height of the BST.

In the worst case $h = O(n)$.

we would like to create tree, whose height is not too large. we want BST to have height $O(\log \frac{n}{2})$

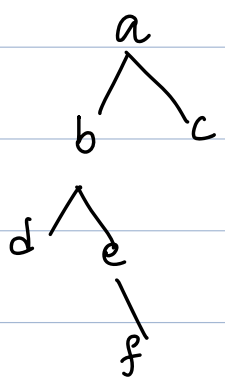
In this topic, we study about AVL trees, which are BST's satisfying some additional constraint (on height)

Remark: In this topic, we use slightly different definition for height of a node in a binary tree.

The height of a node in a binary tree is the number of vertices in the longest Path Connecting that node to a leaf node.

height of a tree = height of the root

Example:



height (b) = 3

height (f) = 1

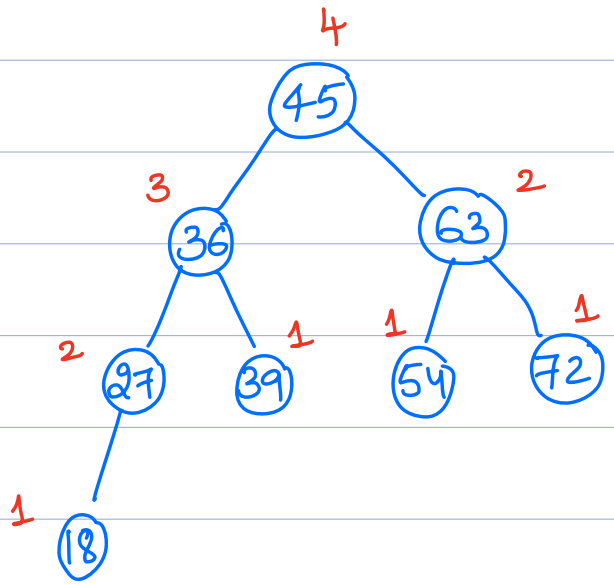
height (a) = 4

height of the tree = 4

AVL Tree

An AVL tree is binary search tree such that for every internal node v of T, the heights of the childrean of v can differ by at most one.

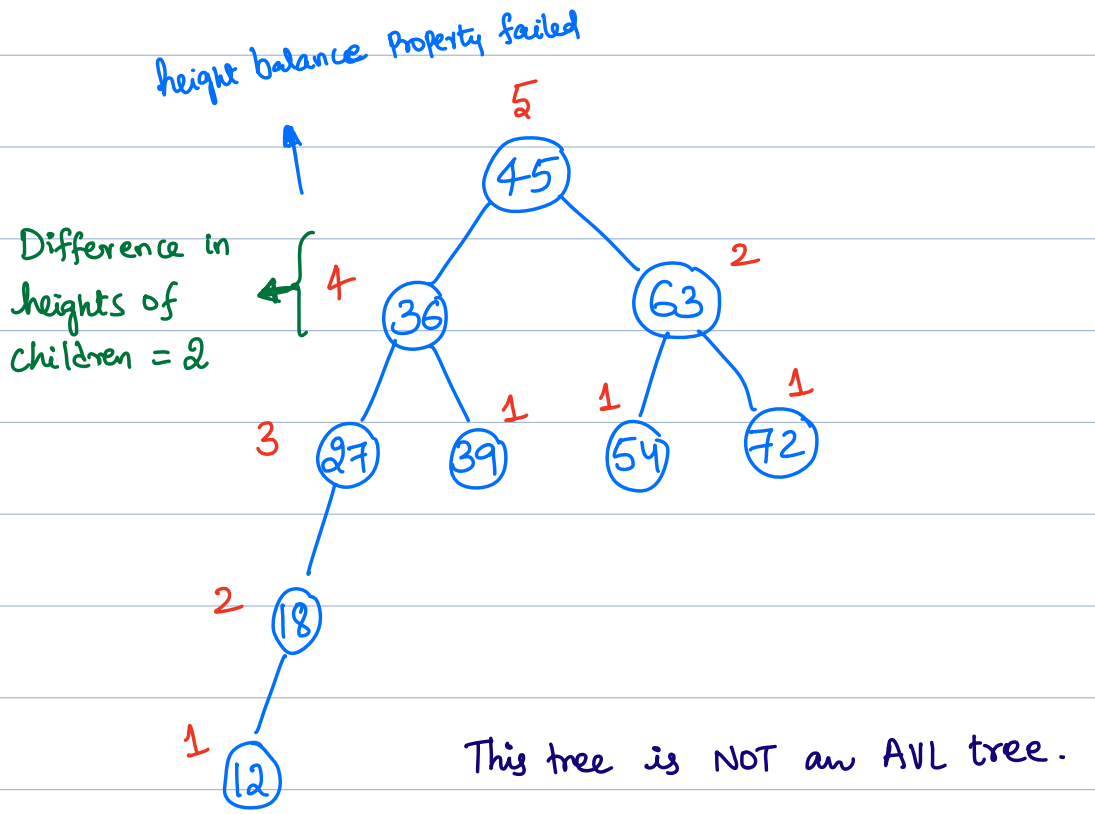
Example:



Example of a AVL-tree: the label on the top of the node denotes the height of the node.

Here we are counting # of vertices.

Note: If a node has no left child | left subtree then then the height of the left subtree is zero.



This tree is NOT an AVL tree.

Def: **AVL balance condition**:

For every node in the tree, the heights of its left subtree and right subtree differ by at most one.

A node v in the tree is **balanced**, if the heights of its left subtree and right subtree differ by at most one.

Representation of AVL trees

- Same as BST, except we need to add a new field to store the height of each node.

```
Struct Node
  {
    int key;
    Struct Node *left;
    Struct Node *right;
    int height;
  };
```

Height of an AVL tree

Let T be an AVL tree having n nodes,

What is the upper bound on height of T?

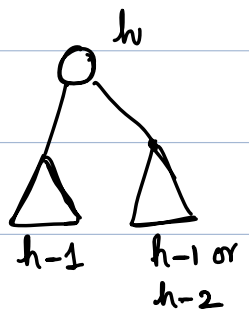
Claim: The height of an AVL tree T having n nodes is $O(\log n)$.

Proof: We first find the minimum number of nodes $n(h)$ in an AVL tree of height h.

It is easy to see that

$n(1) = 1$, $n(2) = 2$ [We are looking for minimum]

For $h \geq 3$, an AVL tree of height h contains the root node, one AVL subtree of height $h-1$ and the other AVL subtree of height $h-1$ or $h-2$.



Since we are looking for minimum # of nodes, the other subtree will have height $h-2$

That is $n(h) = n(h-1) + n(h-2) + 1$ — ①

We know, $n(h-1) \geq n(h-2)$, we get

① becomes

$$n(h) > 2n(h-2)$$
$$> 2 \times 2 \times n(h-4)$$
$$> 8\, n(h-6)$$
$$> 2^i\, n(h-2i)$$

WLOG, h is even

When $i = \frac{h}{2} - 1$,

we get $n(h) > 2^{\frac{h}{2}-1}\, n(2) = 2^{h/2}$

that is $n(h) > 2^{h/2}$

$$h < 2 \log n(h)$$

Since $n(h) < n$

we get $h < 2 \log n$

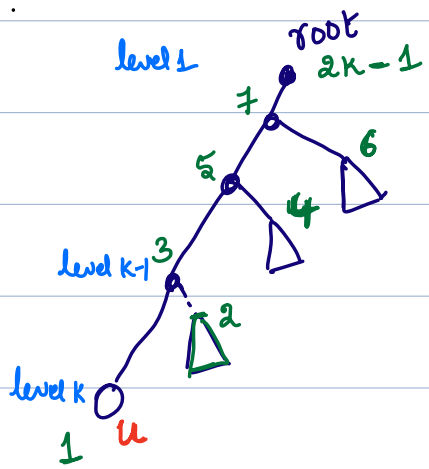
$\therefore$ the height of an AVL tree is $O(\log n)$

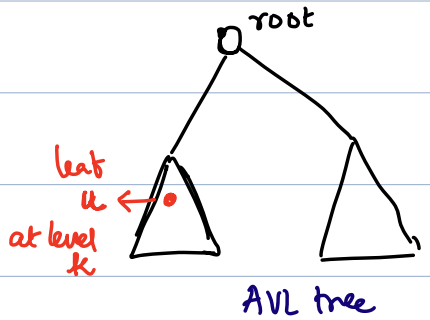
Structure of an AVL tree

Claim: Let T be an AVL tree on n nodes.

If a leaf closest to the root is present at level k, then the height of the tree is at most $2k-1$.

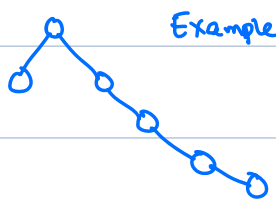
Proof:





Note: The above claim need not hold for arbitrary binary trees.

Example.



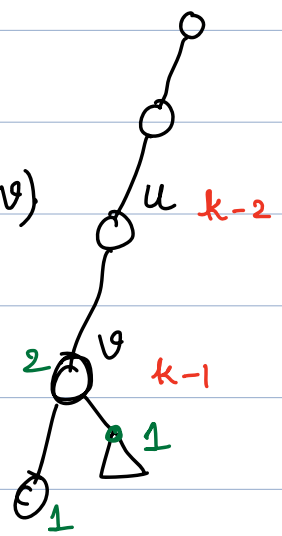
Claim: If a closest leaf is at level k then all nodes at level $1, \ldots, k-2$ have two childrean

Proof: Proof by contradiction

Suppose there is a node u at level $k-2$ that has only one child (Say v)

Note that v can't be a leaf.

that is Subtree rooted v has height two.

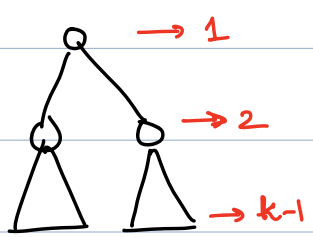


For node u, the height balanced Property is violated at u.

Note: At level $k-1$ nodes may have either one child or two childrean.

From the Previous Claim, we can see that all levels 1 to $k-1$ are full. [Here k denote level of a leaf Closest to the root]

The tree has at least 2^{k-1} nodes



Since the height of the tree is $2k-1$

it can have at most 2^{2k-1} nodes.

ie, $2^{k-1} \leq n \leq 2^{2k-1}$ — (A)

Put h for $2k-1$, then

$$2^{\frac{h-1}{2}} \leq n \leq 2^{h}$$

Note: From the above, we can see that height of an AVL tree is $O(\log n)$, this will improve the running time of basic operations such as INSERTION, DELETION etc.

Remark: From Ⓐ, we can see that if the height of an AVL tree is h, then the leaf closest to the root is at level $\frac{h+1}{2}$.

The first $\frac{h-1}{2}$ levels in AVL tree is Complete binary tree. In Next levels tree may have less nodes.

Story so far

- In an AVL tree of height h, the leaf closest to the root is at level at least $\frac{h+1}{2}$

- On the first $\frac{h-1}{2}$ levels the AVL tree is a complete binary tree

- Number of nodes in the AVL tree is at least $2^{\frac{h-1}{2}}$ and at most 2^h.

Def: A binary tree T is called height balanced if for every node v, height of v's children differ by atmost one.

Operations

- SEARCH — EASY, as AVL tree a BST of height $O(\log n)$
- INSERTION
- DELETION

INSERTION

The insertion routine for AVL trees starts exactly the same as the insertion routine for BST's,

But after the insertion of the node in a subtree, we must check whether the subtree has become unbalanced. If so, then we need to rebalance it.

That is

Inserting a node into AVL tree changes the height of some of the nodes in T

Q: What are the nodes whose height could change after inserting a node v?

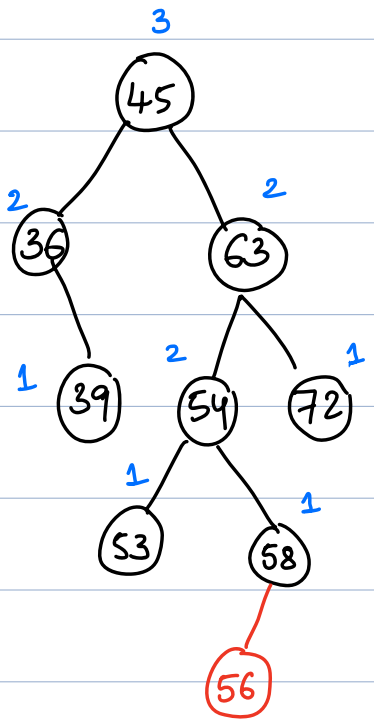
Ans: ancestors of v.

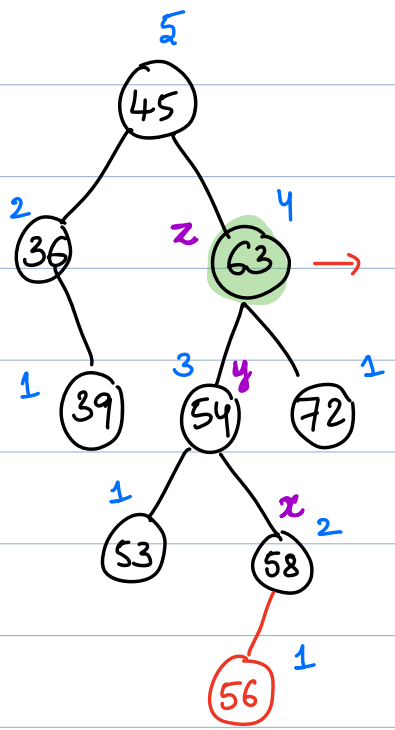
- If insertion causes T to become unbalanced, then some ancestor of v would have a height imbalance.

- We travel up the tree from the newly created node untill we find the first node x such that its grandParent z is unbalanced node.

  let y be the Parent of node x.

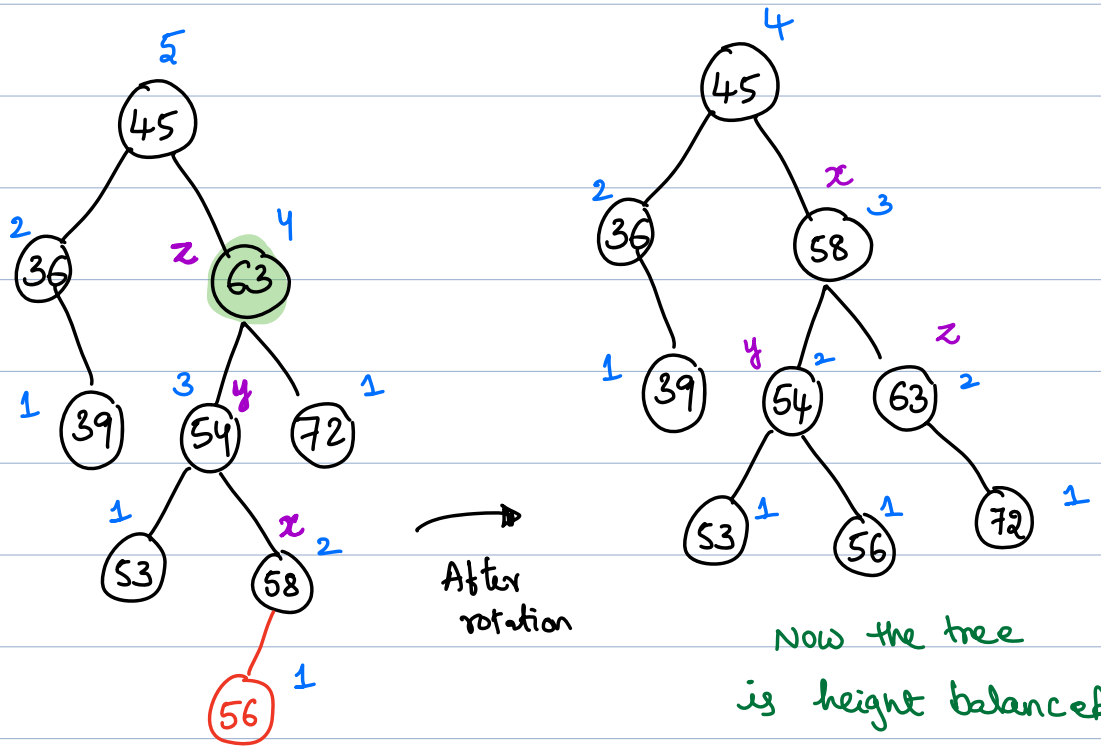
We insert the node 56.





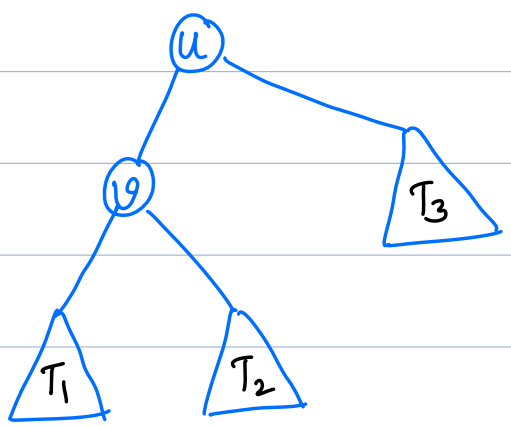
→ height balance Property Voilated.

Idea: We will rotate the tree so that height balance Property restored.


5
45
2
36
z
4
63
1
39
3
y
54
1
72
1
53
x
2
58
1
56
After rotation
4
45
2
36
x
3
58
1
39
y
2
54
z
2
63
1
53
1
56
1
72
Now the tree is height balanced

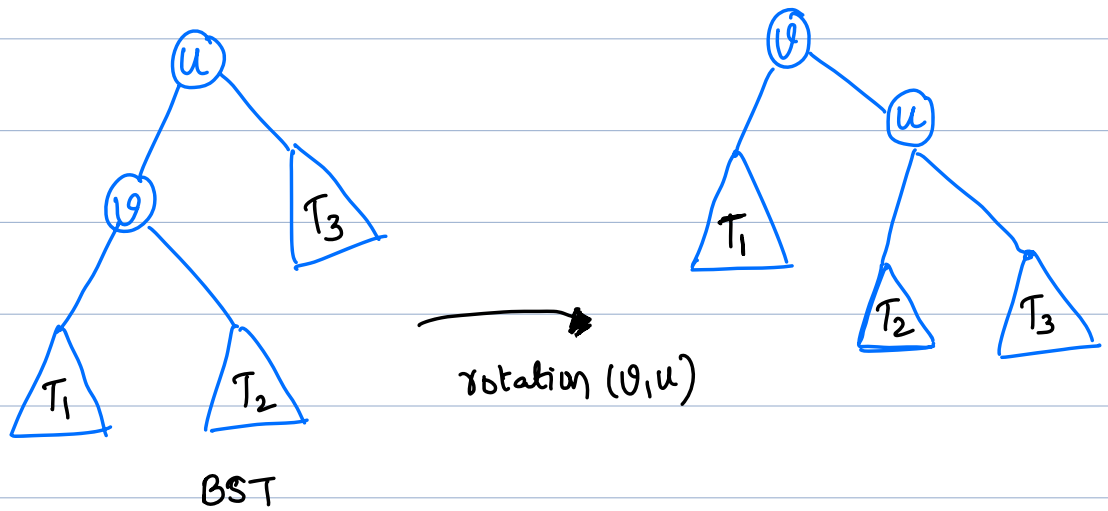
Def: Rotation is a way of locally reorganizing a BST.

let u & v be two nodes such that u is the Parent of v.



BST

$$keys(T_1) < key(v) < keys(T_2) < key(u) < keys(T_3)$$



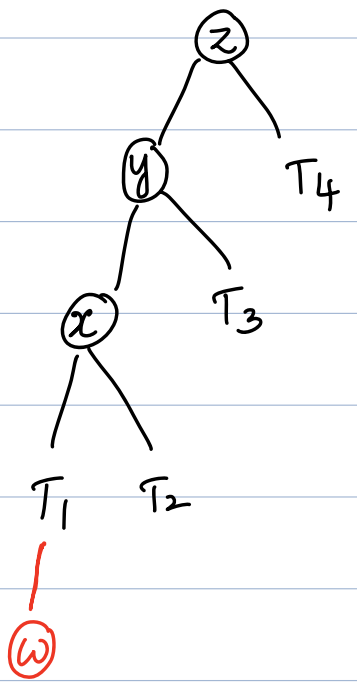
BST

INSERTION

Suppose insertion happens in subtree T_1

Let z be the first imbalance node after insertion.

ie, x & y are balanced after insertion of w.



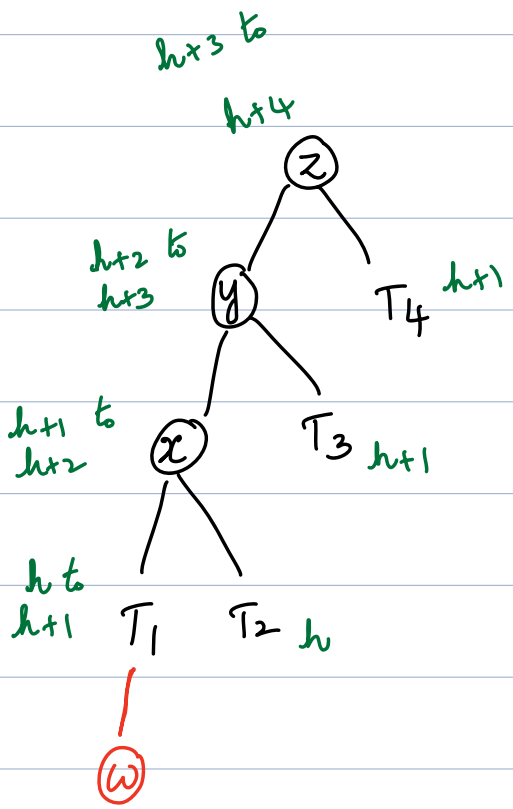
height (T_1) increases from h to $h+1$.

- Since x remains balanced after insertion of w.

the height (T_2) is h or $h+1$ or $h+2$

- if height $(T_2) = h+2$ then original tree was unbalanced
- if height $(T_2) = h+1$ then height (x) does not increase
- so height $(T_2) = h$
- height (x) increases from $h+1$ to $h+2$.

Continuing above arguments



Since y remains balanced

height (T_3) = h+1 or h+2 or h+3

- if height (T_3) = h+3 then

  y originally unbalanced

- if height (T_3) = h+2 then

  height of y does not increase

So height (T_3) = h+1

then height (y) = h+3

- Note that z is imbalanced after insertion.

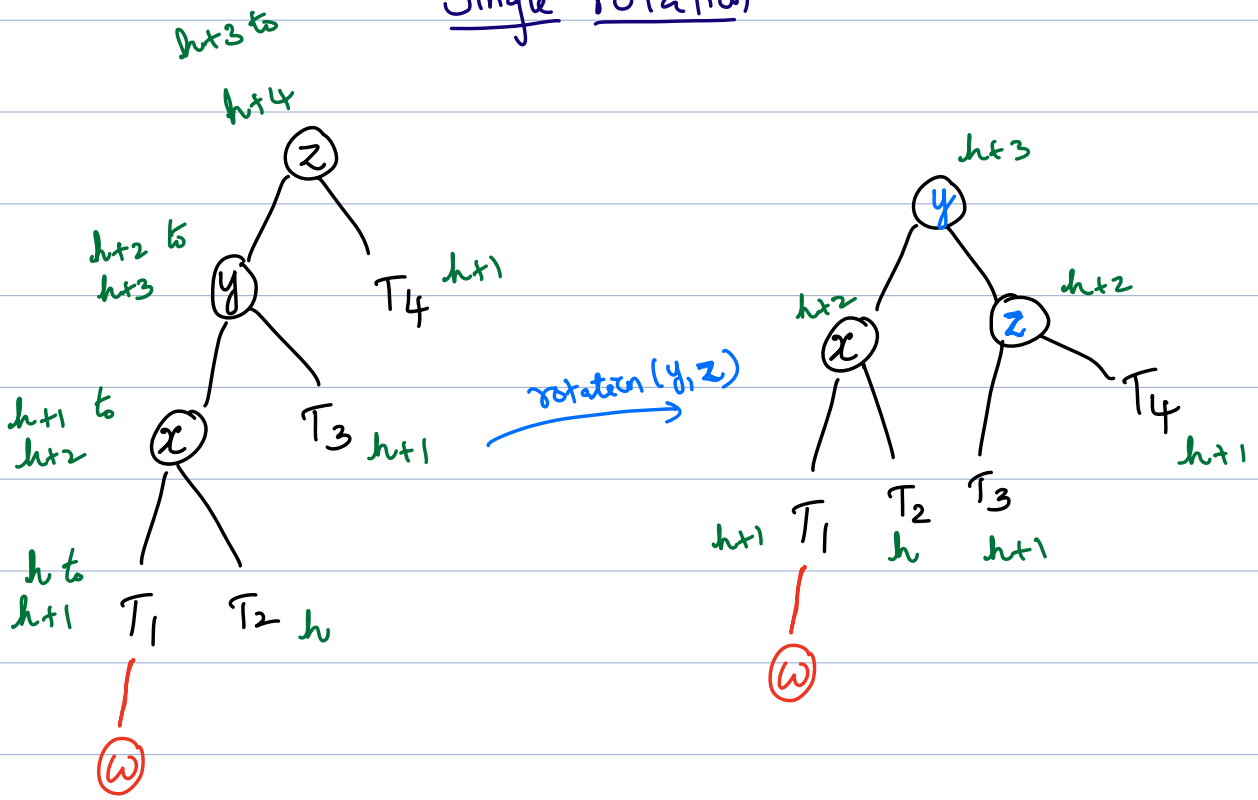
Since z was balanced originally

height (T_4) = h+1, h+2, h+3

Now z is imbalanced

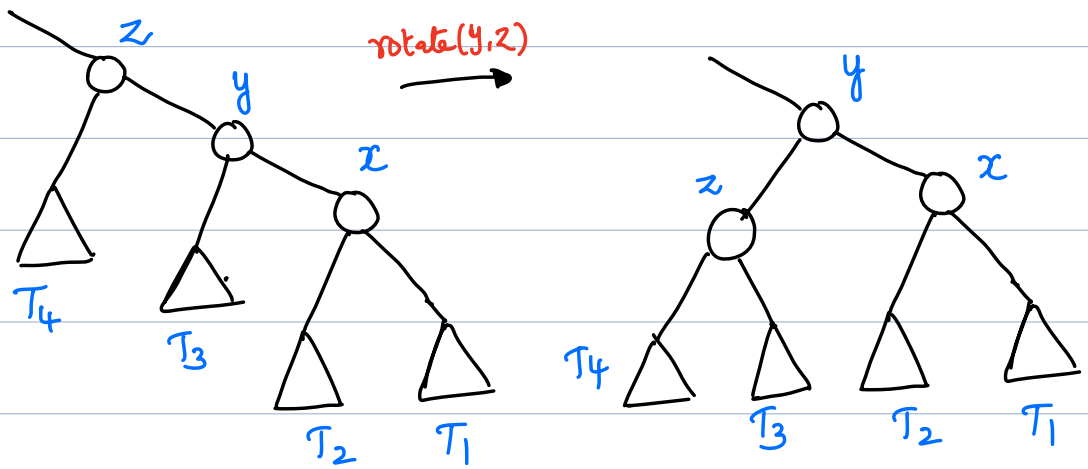
height (T_4) = h+1

Single rotation

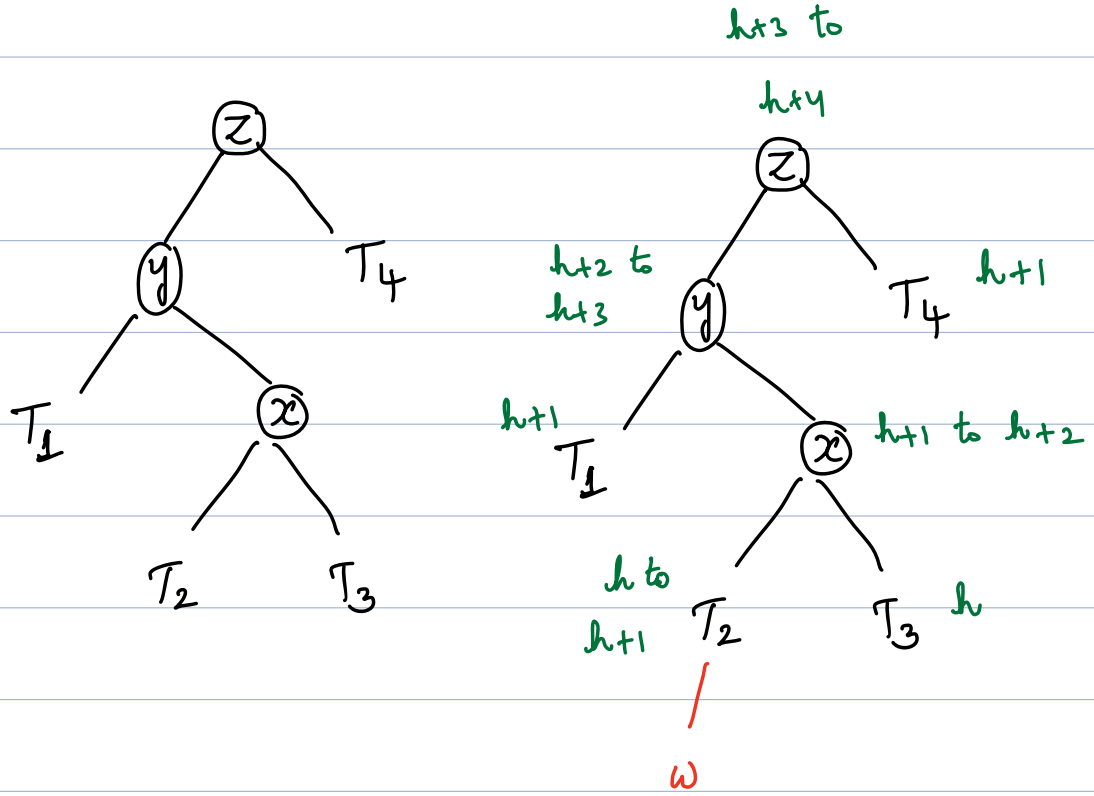


Remark: The height of the subtree remains the same after rotation. Hence no further rotations required.

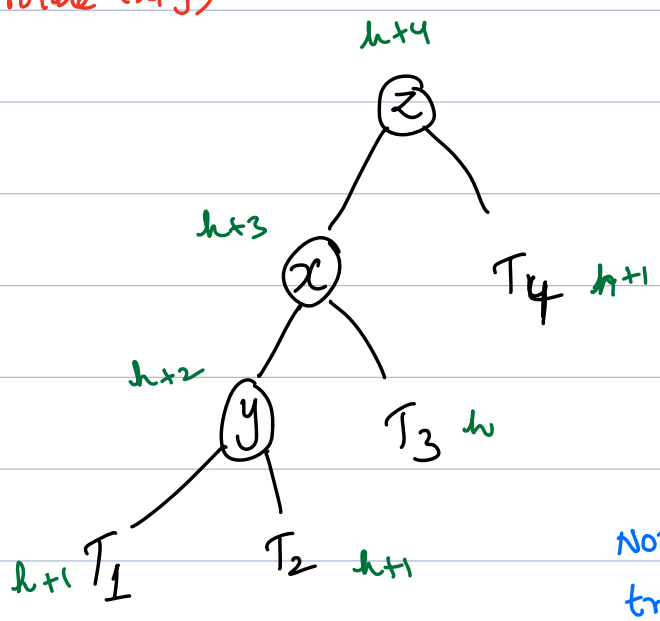
Single rotation : Symmetric case



Double rotation

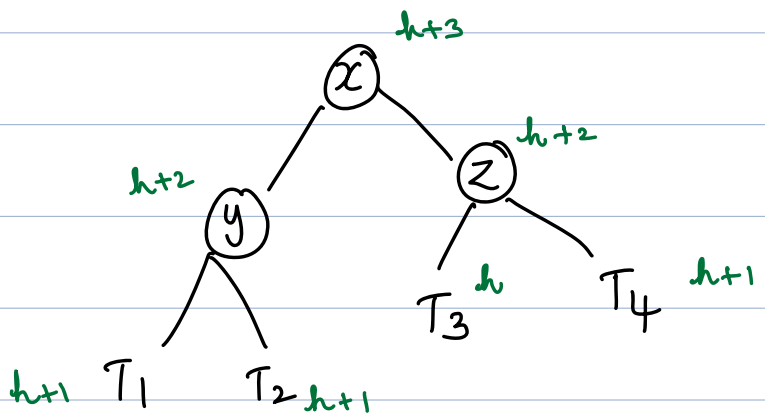


First we rotate (x, y)



Note that Still the tree is not balanced.

Next rotate (x, z)

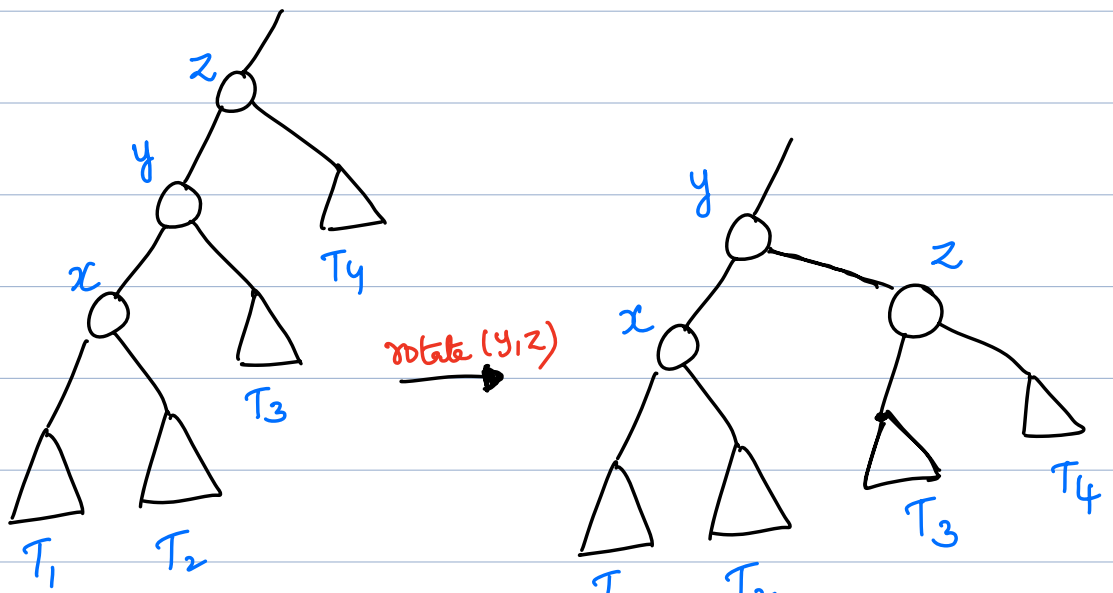


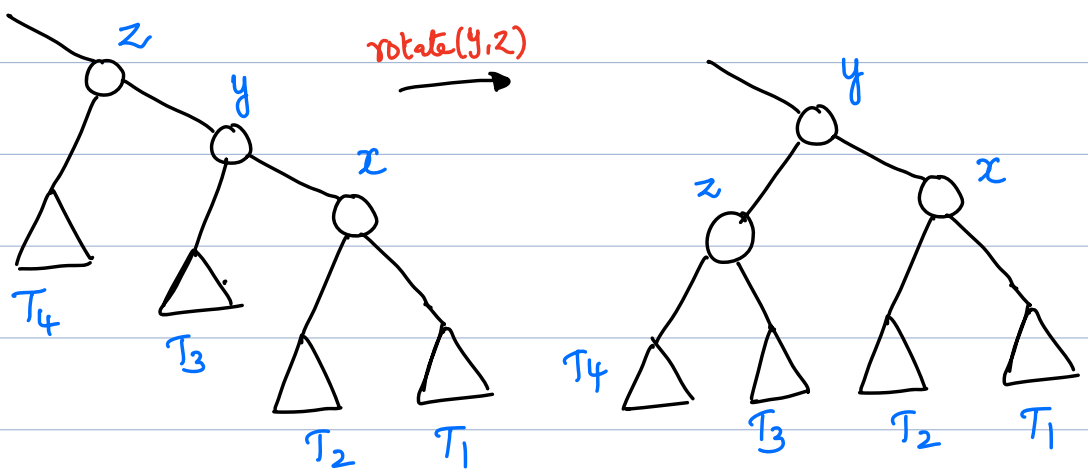
Since the final tree has same height as original tree. Hence we need not go further up the tree.

Summary - Sofar

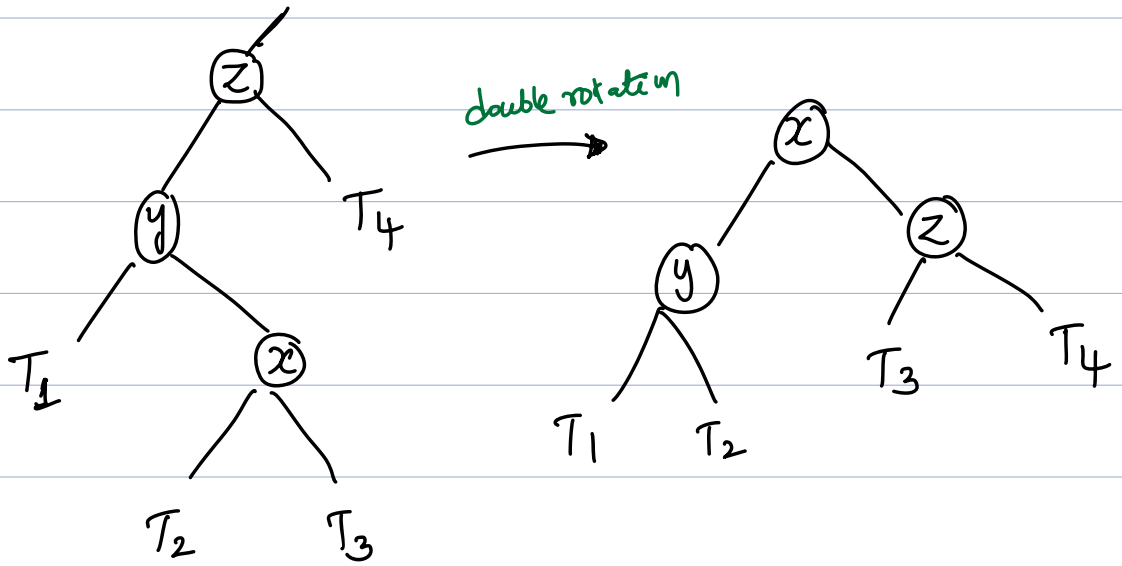
There are four ways to rotate nodes in an AVL tree.

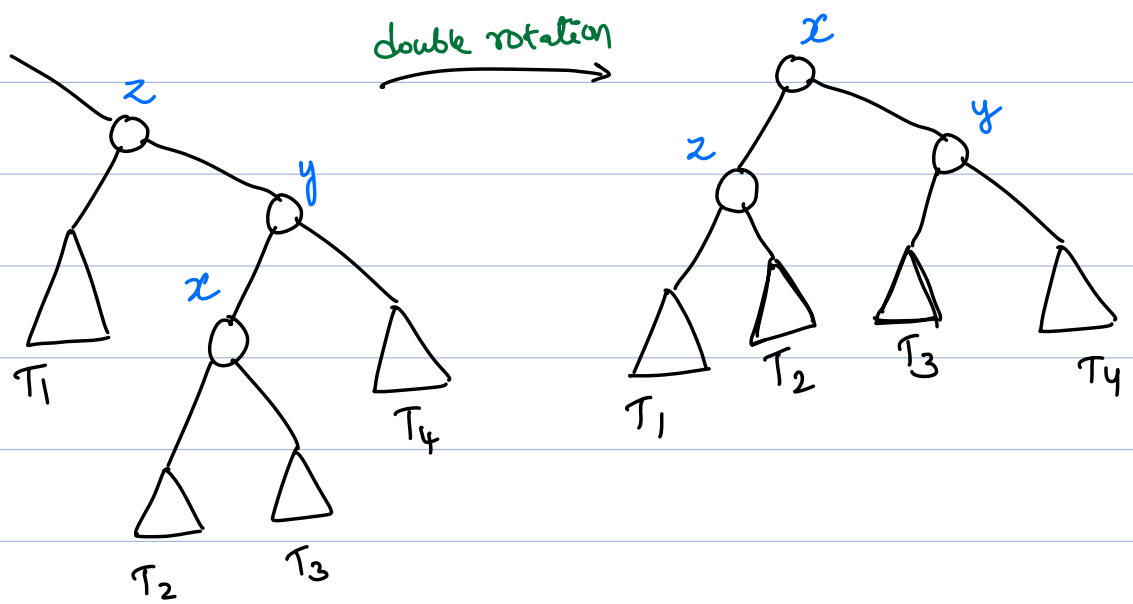
- Single Rotations





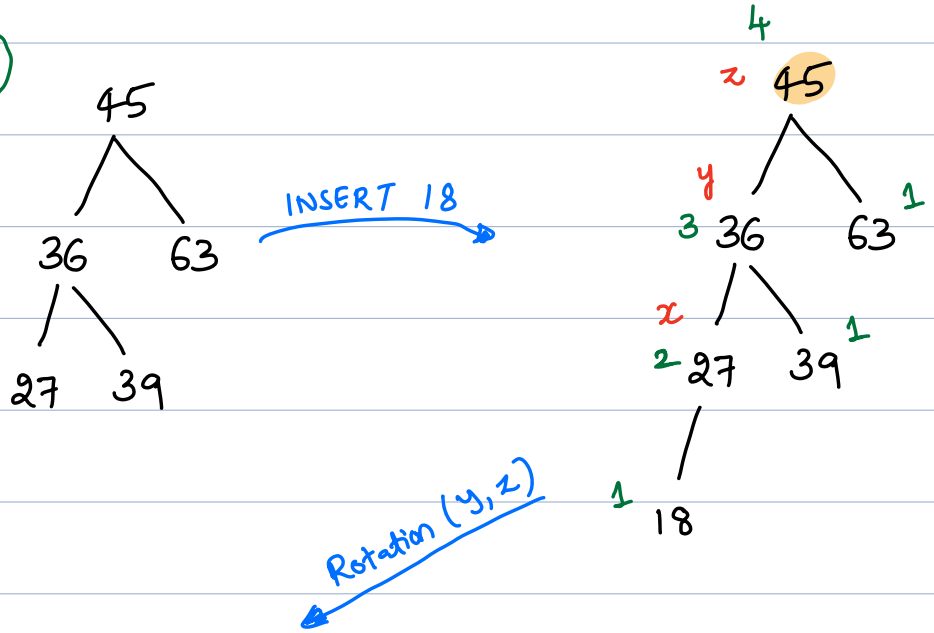
– Double Rotations

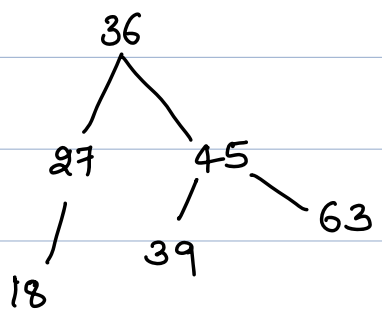




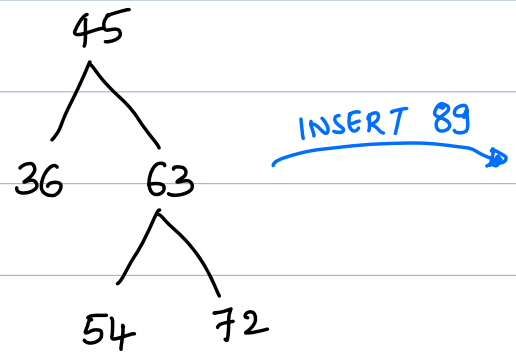
Examples [All four cases]

(I)

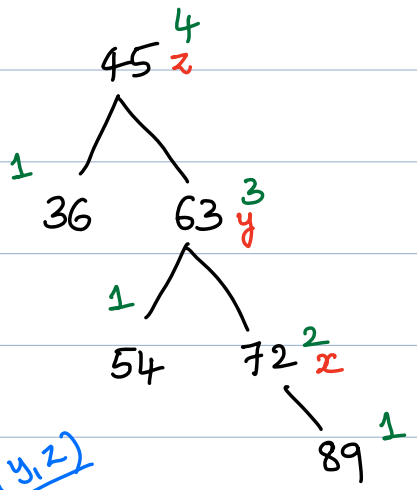




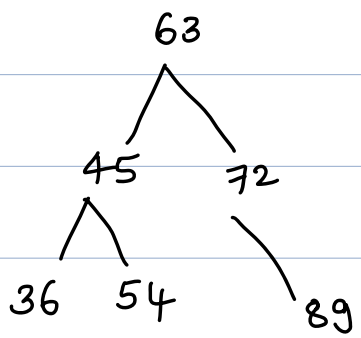
II



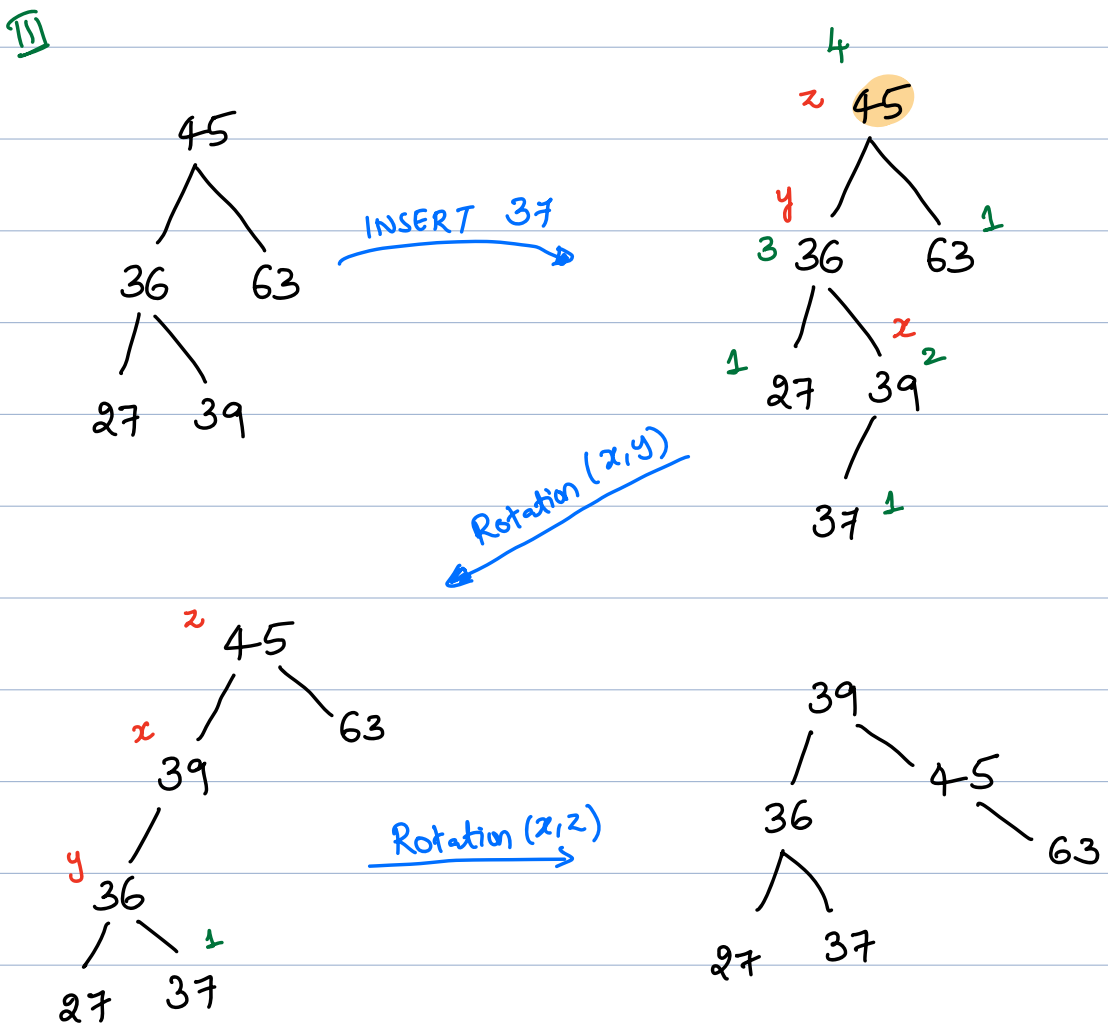
INSERT 89



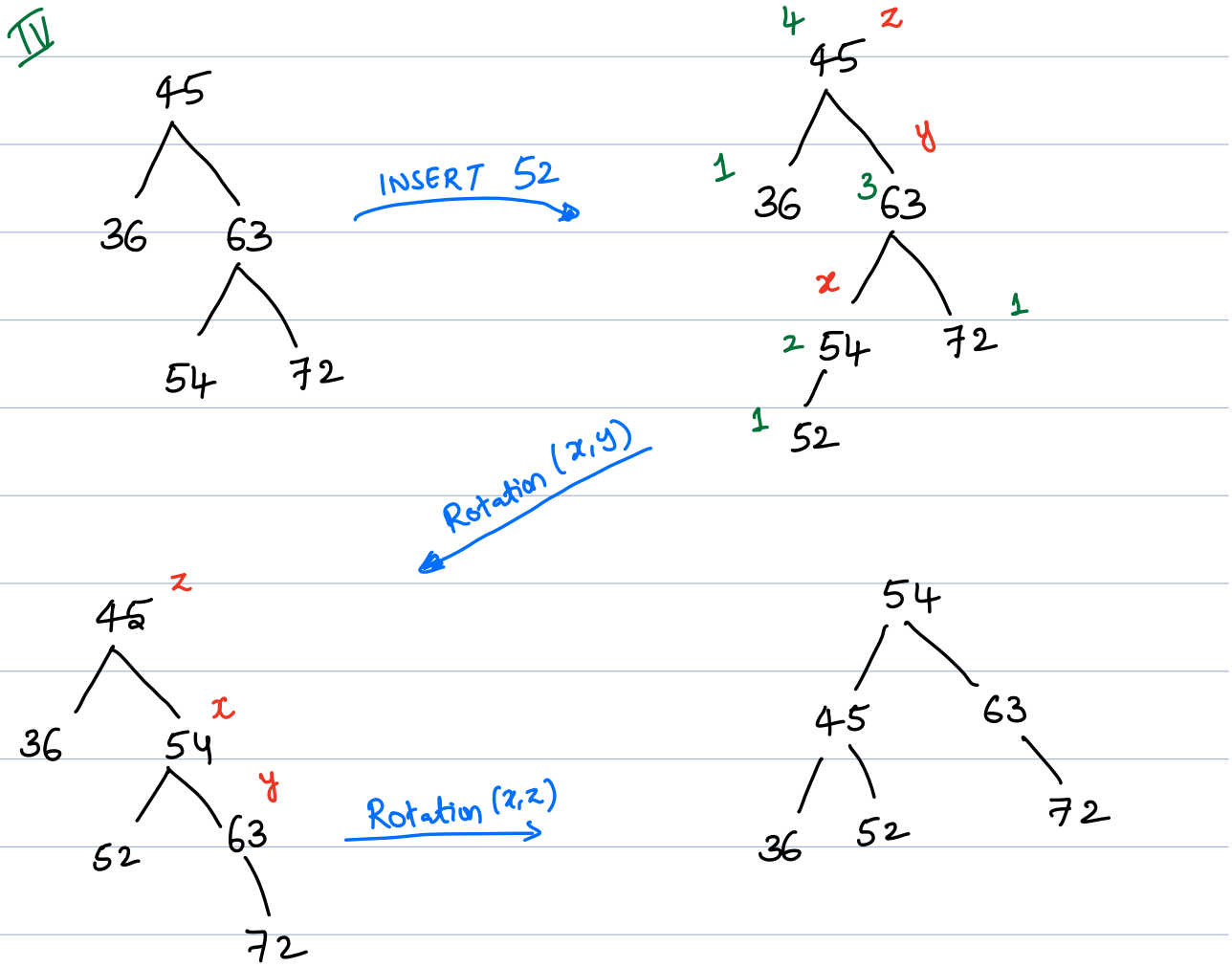
Rotation (y, z)



11


45
36
63
27
39
INSERT 37
4
z
45
y
3
36
1
63
1
x
2
27
39
37
1
Rotation (x,y)
z
45
x
39
63
y
36
1
27
37
Rotation (x,z)
39
36
45
27
37
63

IV


45
36
63
54
72
INSERT 52
4
z
45
y
1
36
3
63
x
1
2
54
72
1
52
Rotation (x,y)
z
45
36
x
54
y
52
63
72
Rotation (x,z)
54
45
63
36
52
72

Deletion

Recap

When deleting a node in a BST, we either delete a leaf or a node with only one child.

True | False :

In an AVL tree if a node has only one child, then that child is a leaf.

Ans: True

if that child is not leaf then its Parent height is NOT balanced.

- So, in an AVL tree we either delete a leaf or the Parent of a leaf
- We can simply think about it as deleting a leaf.

Issue: After deletion the tree may be unbalanced.

[The height of ancestors of w may reduce]

Let w be the node deleted and

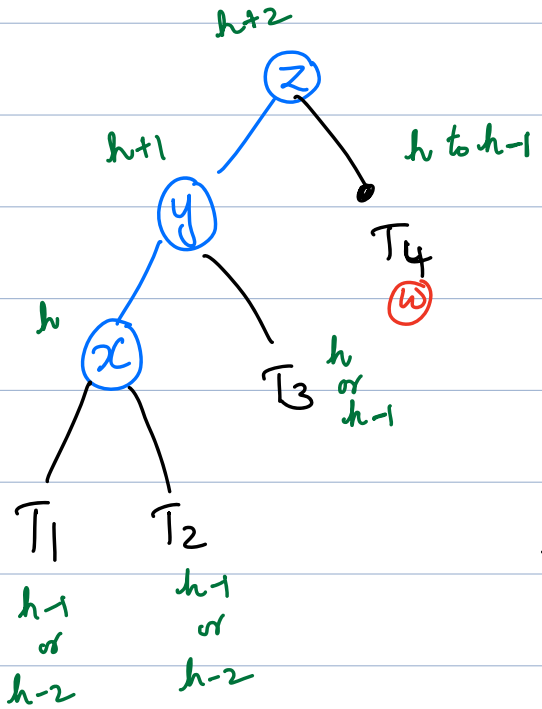
Let z be the first unbalanced node encountered while travelling up the tree from w.

Let y be the child of z with larger height &

Let x be the child of y with larger height

Idea is to perform rotations to restore balance at the subtree rooted at z.

Note: Unlike the case of INSERTION, the rotation operations may upset the balance of another node higher in the tree, we will continue checking for balance untill the root of T is reached.

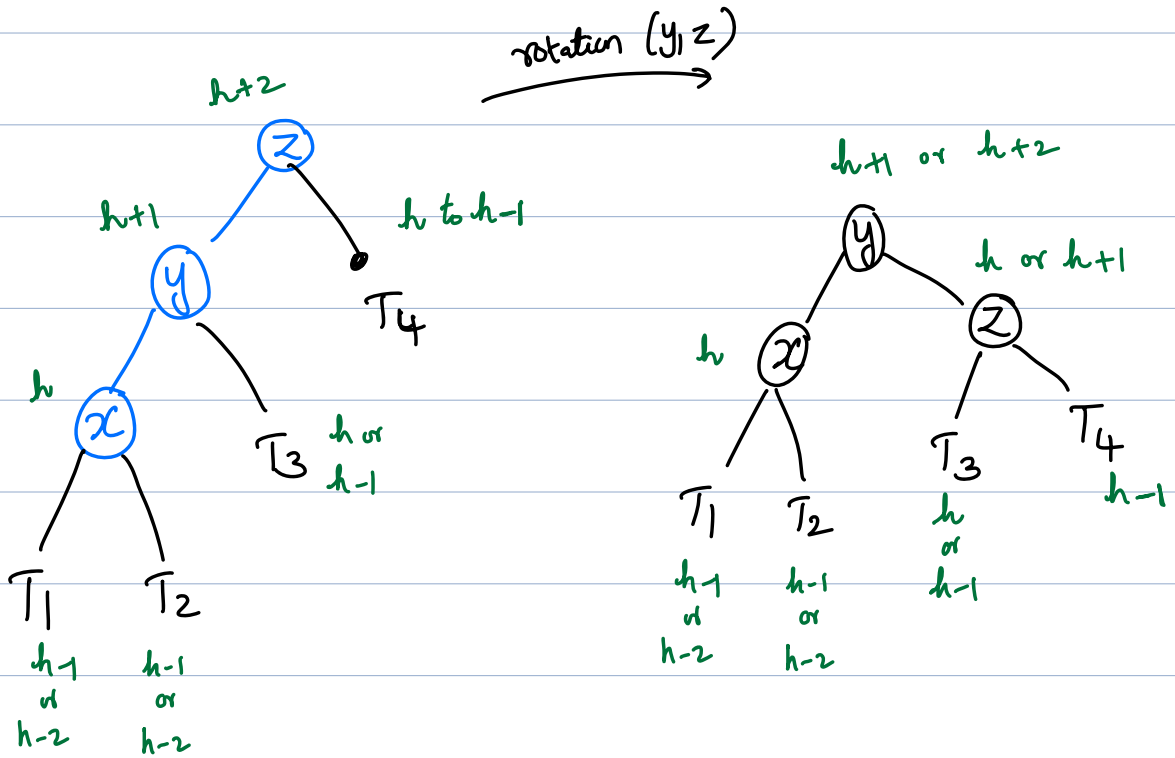


- Suppose deletion happens in subtree T_4 and its height reduces from h to $h-1$

[Q: Why y is not in T_4]

- Both T_1 & T_2 cannot have height $h-2$.

Deletion - Single rotation



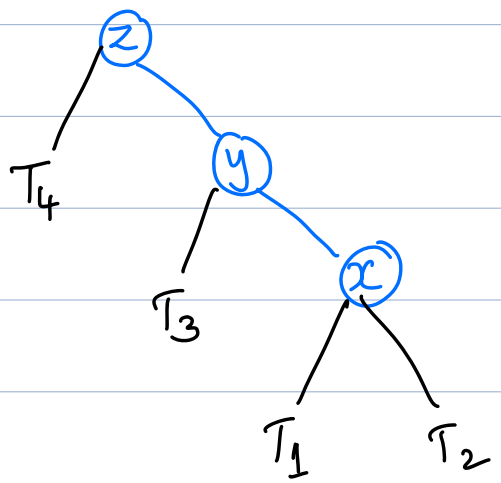
IMP Note: If the height y after rotation is $h+2$, which is same height of the original subtree, then we are done.

If the height y after rotation is $h+1$ then the subtree rooted y height is reduced, then we have continue up the tree.

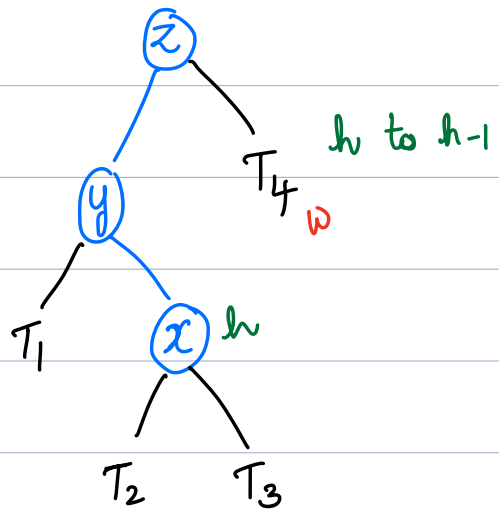
[we look for new z, y, and x]

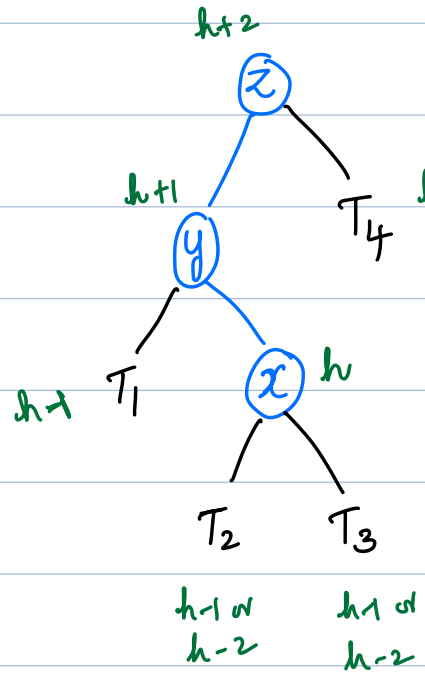
ie, the whole subtree rooted y can be treated as T_4 (in the previous case)

Case-II : Symmetric to case-I [details are skipped]



CASE - III [Double Rotation]



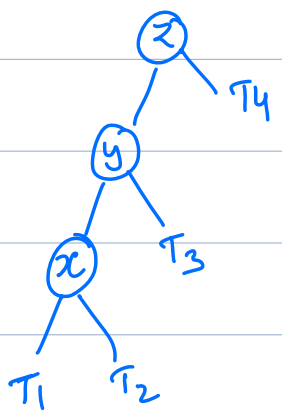


- Since y is balanced height T_1 is h or h-1

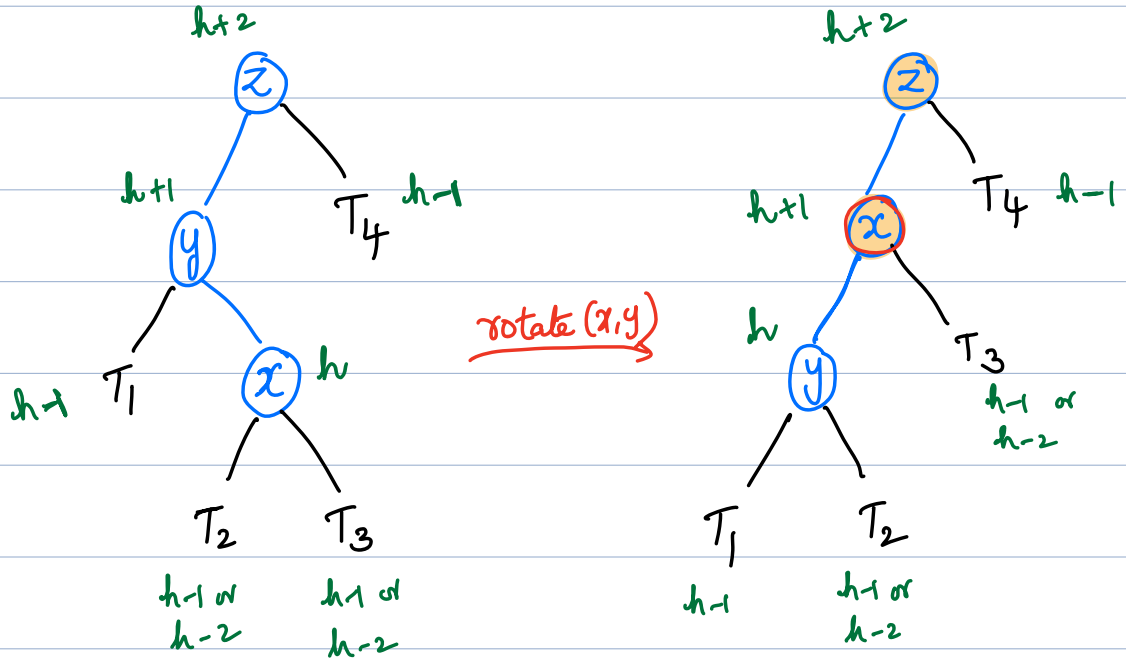
IMP

- If height T_1 is h then we will select root of T_1 as x, so that we can reduce to the Previous case

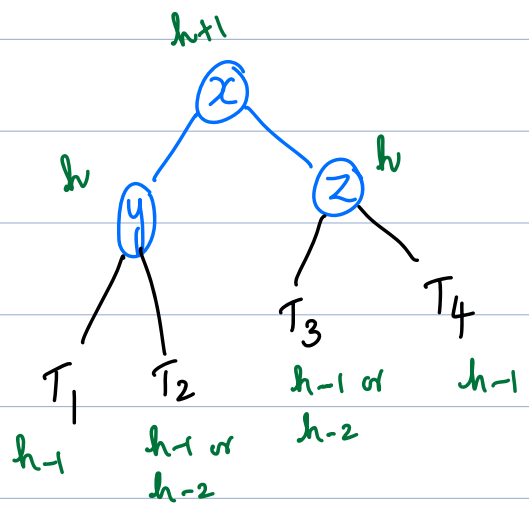
ie.,



∴ we assume height T_1 is h-1.



There could be imbalance at x.



Remark: After rotations, the final tree has height less than original tree. Hence we need to continue up the tree.

Running time of insertion & Deletion

Both takes time $O(\log n)$

- In case of insertion, we might have to go $O(\log n)$ levels to find the unbalanced node.

- In case of deletion, we need $O(\log n)$ to delete a node & for rebalancing.