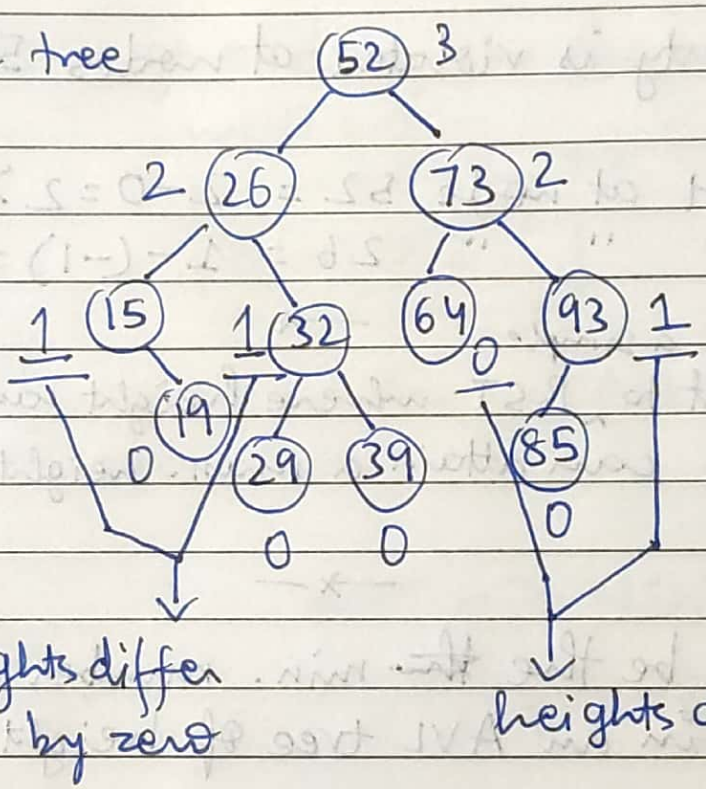
AVL Tree

AVL tree is named after 3 computer scientists:- Adelson, Velskii & Landis

It is a BST with the following property.

For every node in the tree, the heights of the children can differ by atmost 1.

Ex:- AVL tree



The no. written beside each node = height of the node

heights differ by zero

heights differ by one

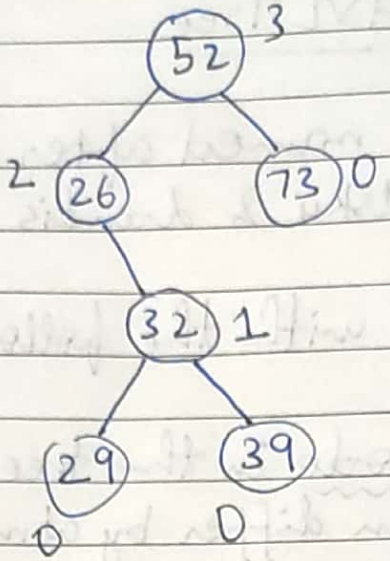
height of a NULL-child is counted as '-1'.

⇒ in the above example the heights for the children of node - (15) are:-

-1 & 0

Hence, the diff. = 0 - (-1) = 1

Ex:→ NOT an AVL tree



diff. in height (DIH)

AVL property is violated at nodes 52 & 26 resp:-

DIH at node 52 = 2-0 = 2 > 1

" " " 26 = 1-(-1) = 2 > 1

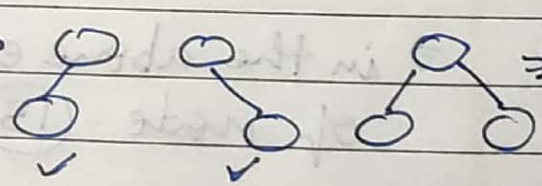
→ ✗

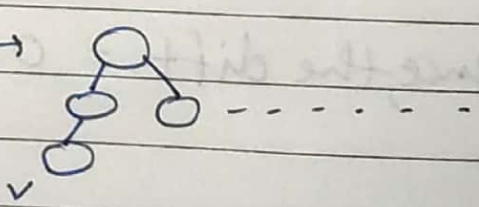
* → In contrast to a simple BST where height can go upto n-1, AVL tree can attain a max. height upto 2 log n.

→ ✗ —

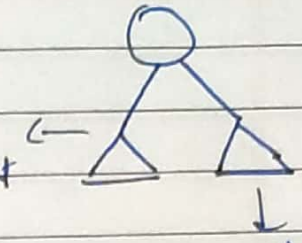
Proof → Let N(h) be the ~~th~~ min. number of nodes required in an AVL tree of height h.

AVL trees of height 0 →  ⇒ N(0) = 1

AVL trees of height 1 →  ⇒ N(1) = 2

AVL trees of height 2 → 

⇒ N(2) = 4



$\Rightarrow N(h) = N(h-1) + N(h-2) + 1$

$> N(h-1) + N(h-2)$

$> N(h-2) + N(h-2)$

$= 2N(h-2)$

If we go on doing like this we'll get :-

$$N(h) > 2^{h/2} \quad \star\star$$

Now, let us consider an AVL tree of height h & n nodes. Then

$$n \geq N(h) > 2^{h/2}$$

$$\Rightarrow h < 2 \log n$$

i.e. h is $O(\log n)$ (Q.E.D)

$\star\star \rightarrow$ The height of an AVL Tree is $O(\log n)$

∴ AVL tree is a BBST.

(*) If the AVL tree property is violated at a node, then the height of the node must be atleast 2.

balanced here means a tree which satisfies AVL property.

We use rotation operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation like insertion.

⇒ ROTATION

H(z) denotes height of node z

Let z be a node where the AVL property is violated.

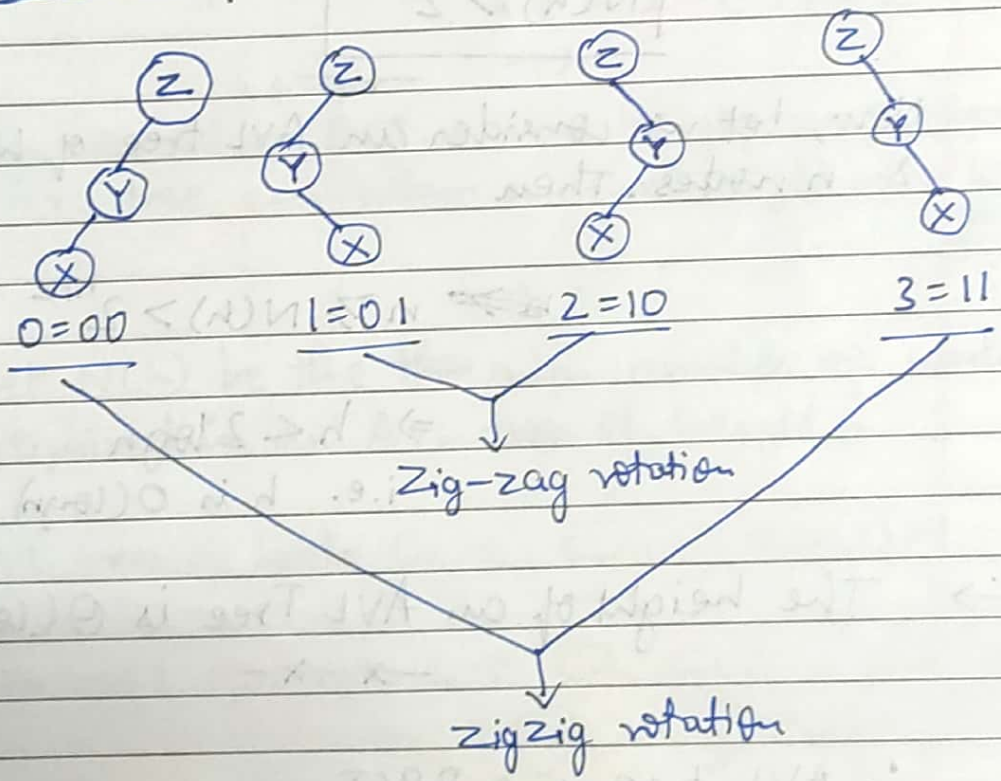
Y be the child of z such that H(z) = H(Y) + 1
X " " " " Y " " H(Y) = H(X) + 1

We ~~dont~~ already know that H(z) > 1 (last point in prev. pg)

Hence X & Y are valid nodes in the AVL tree.

— x —

(*) 4 Rotations



To know which kind of rotation is it.
You'll travel down from z, maintain a variable which gets updated when you move right or left.

For ex let the variable is $c=0$

Take 0 when left & 1 when right

~~Case0~~ we start from z.

Case0 $z \xrightarrow[c=0\times2]{left} y \xrightarrow[c=0+(0\times1)=0]{left} x \Rightarrow c=0 \Rightarrow$ ZigZig(0) rotation

Case1 $z \xrightarrow[c=0\times2]{left} y \xrightarrow[c=0+(1\times1)=1]{right} x \Rightarrow c=1 \Rightarrow$ ZigZag(1) rotation

Case2 $z \xrightarrow[c=1\times2]{right} y \xrightarrow[c=2+(0\times1)=2]{left} x \Rightarrow c=2 \Rightarrow$ ZigZag(2) rotation

Case3 $z \xrightarrow[c=1\times2]{right} y \xrightarrow[c=2+(1\times1)=3]{right} x \Rightarrow c=3 \Rightarrow$ ZigZig(3) rotation

—x—

→ ZigZig rotation takes place by replacing the subtree of z with subtree of y.

→ ZigZag rotation takes place by replacing the subtree of z with subtree of x.

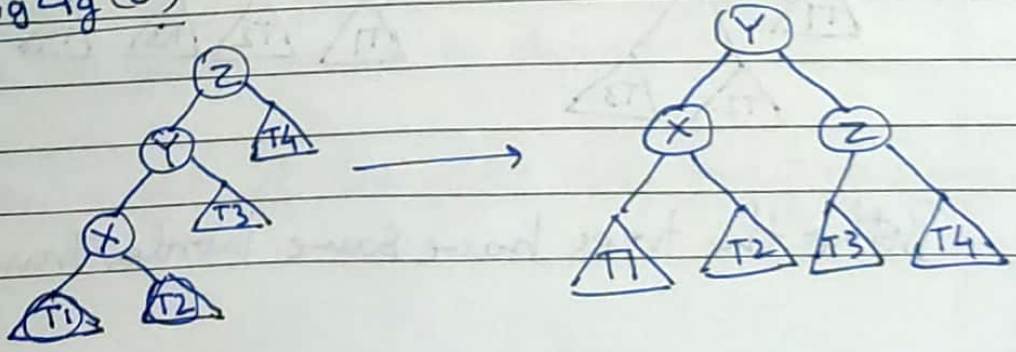
—x—

How each rotation takes place is shown precisely in the pictures as follows:-

○ → represents nodes

△ → denotes subtrees

1.) ZigZig(0)



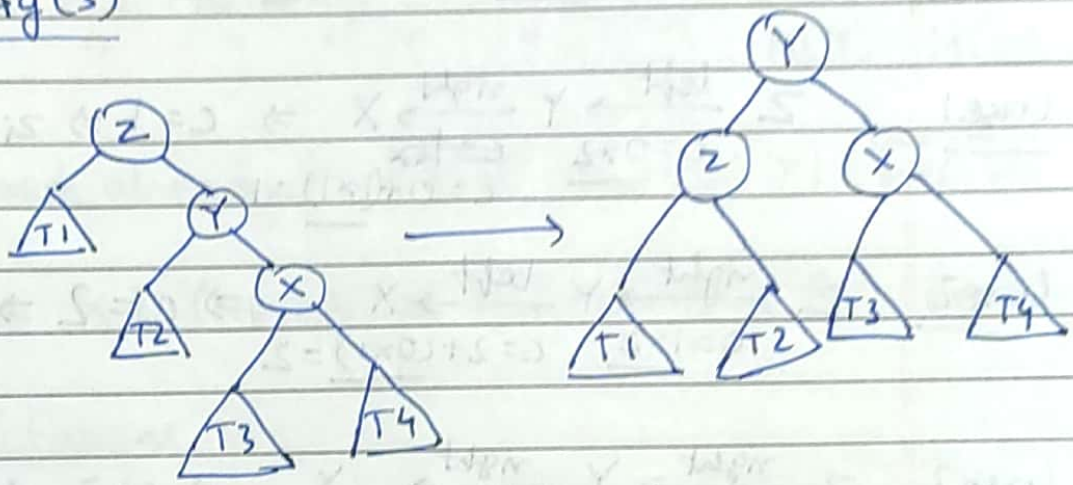
Observe that both trees have same inorder traversal

simple code →

```
T3 = Y→right ; Y→right = Z ; Z→left = T3 ;
Y→p = Z→p ; Z→p = Y ; T3→p = Z ;
```

—x—

2.) ZigZig (3)



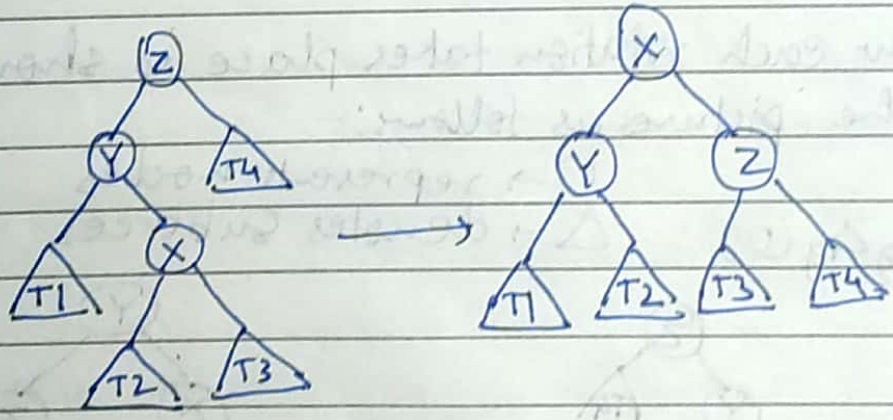
Both the trees have same inorder traversal

Simple code:-

```
T2 = Y→left ; Y→left = Z ; Z→right = T2 ;
Y→p = Z→p ; Z→p = Y ; T2→p = Z ;
```

—x—

3.) ZigZag (1)



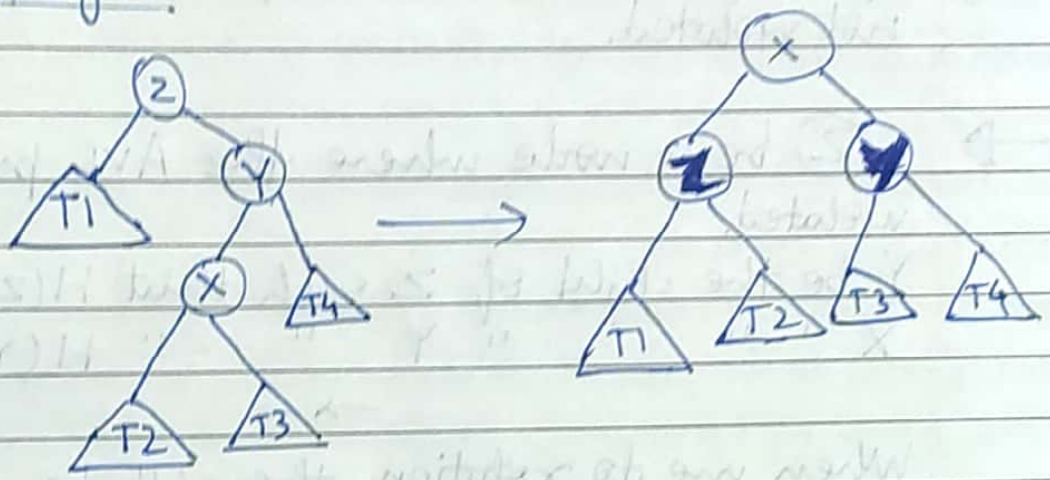
Bothe the trees have same inorder traversal

Simple code:-

```
T2 = X→left; T3 = X→right; Y→right = T2;
Z→left = T3; X→p = Z→p; Z→p = X; Y→p = X;
T2→p = Y; T3→p = Z; X→left = Y; X→right = Z
```

—x—

4.) Zig2ag (2)



Inorder traversals of both trees are same.

Simple code:-

```
T2 = X→left; T3 = X→right; Y→left = T3; Z→right = T2;
X→p = Z→p; Z→p = X; Y→p = X; T2→p = Z; T3→p = Y;
X→left = Z; X→right = Y;
```

—x—

(*) For Deletion, the rotation operations will only be applied for cases 0 & 1
b/c in case 2
we only replace the key of the node by the value of its predecessor. So there is no change for height of node to change.

The heights of the node can only change in case 0 & 1

(*) Go to the parent of the deleted node

1.) check if the AVL property is violated.
2.) update the height

STOP : if height does not change & AVL property is not violated.

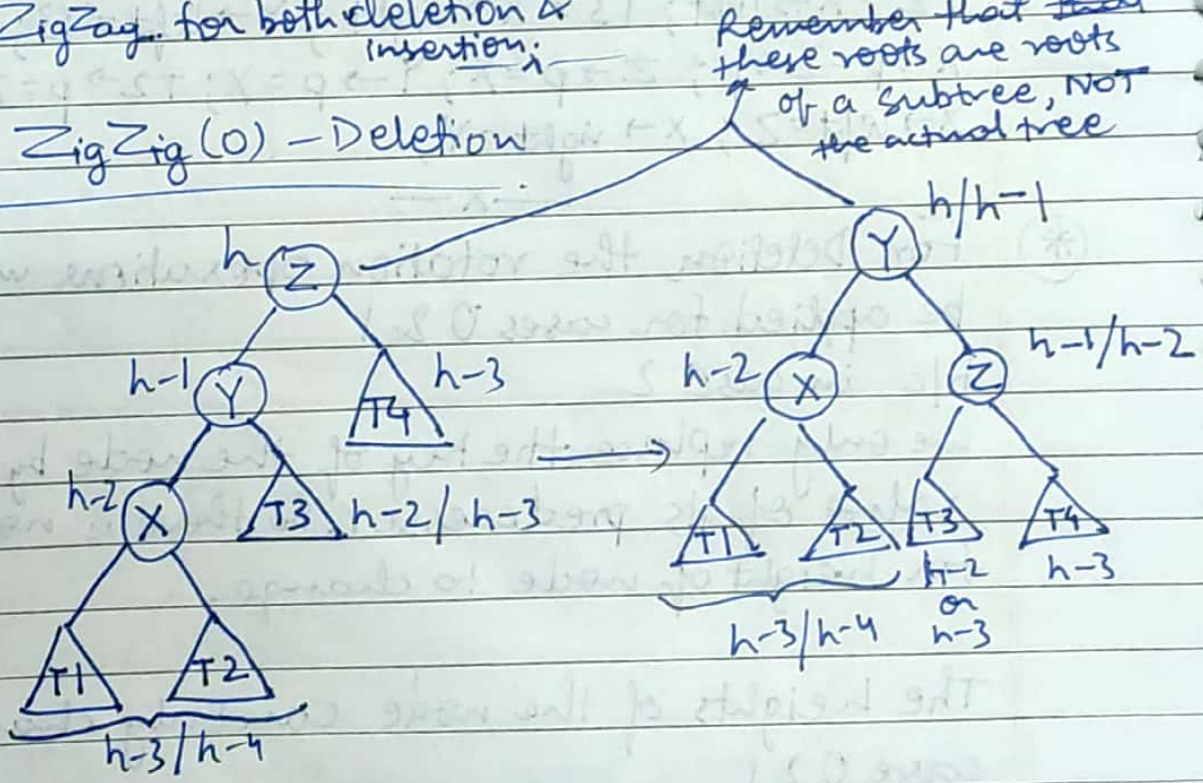
→ Let (Z) be a node where the AVL property is violated.
Y be the child of Z such that H(Z) = H(Y)+1
X " " " " Y " " H(Y) = H(X)+1

—x—

When we do rotation, there'll be 2 symmetric cases of ZigZig & 2 symmetric cases of ZigZag.

So we'll see one case of ZigZig & one case of ZigZag. for both deletion & insertion.

⇒ ZigZig (O) – Deletion



∴ The ⊙ AVL property is violated at Z only, thus the above pic. It is easy to ⊙bserve that deletion of node happened in

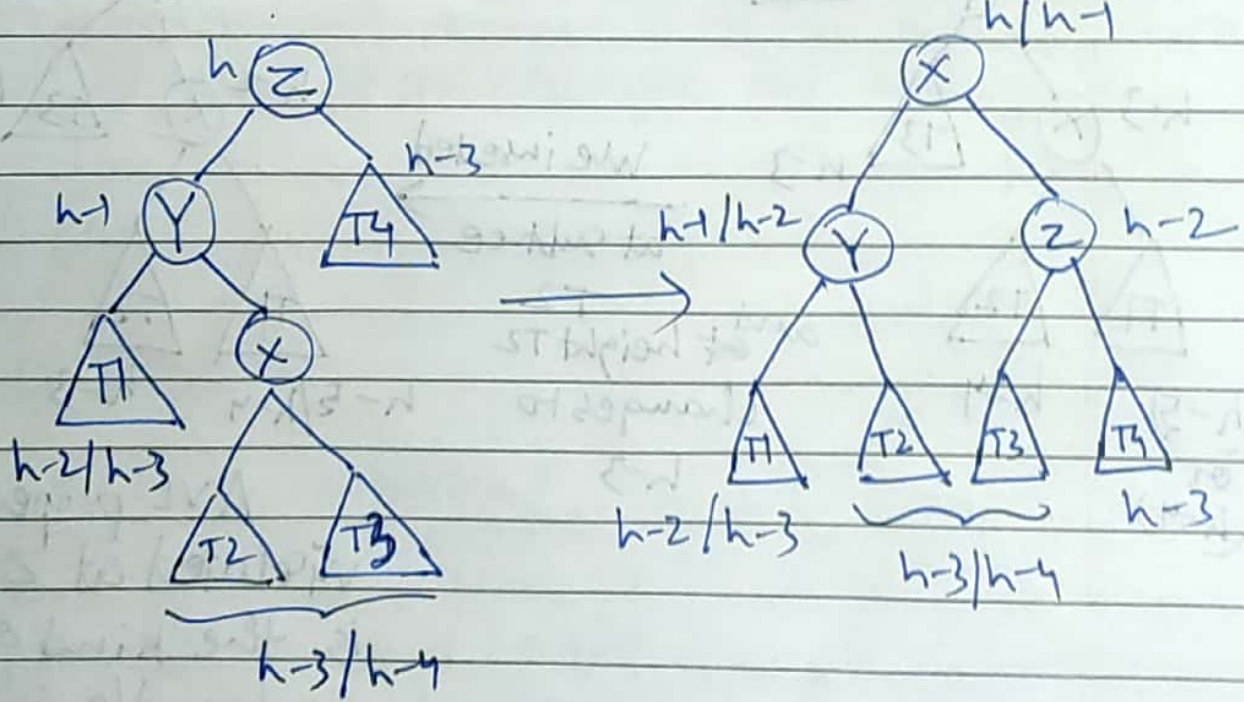
T 4.

* Keep in mind that heights of the subtrees T1, T2, T3 & T4 won't change b/c of rotation.

We see that the root changes from Z to Y the height of Z was h, while the height of Y can be h or h-1. If the height of Y becomes h-1 i.e. if the height of root changes, there may be a violation of AVL prop. at the parent of the root. So, we'll check the parent of Y (root).
∴ These no. of rotations can be of order H = log n.
However, it's crucial to see that the present subtree now satisfies AVL property (after rotation).

~~The same things will apply to ZigZag case.~~

⇒ Zig Zag (L) - deletion



Insert in AVL Tree

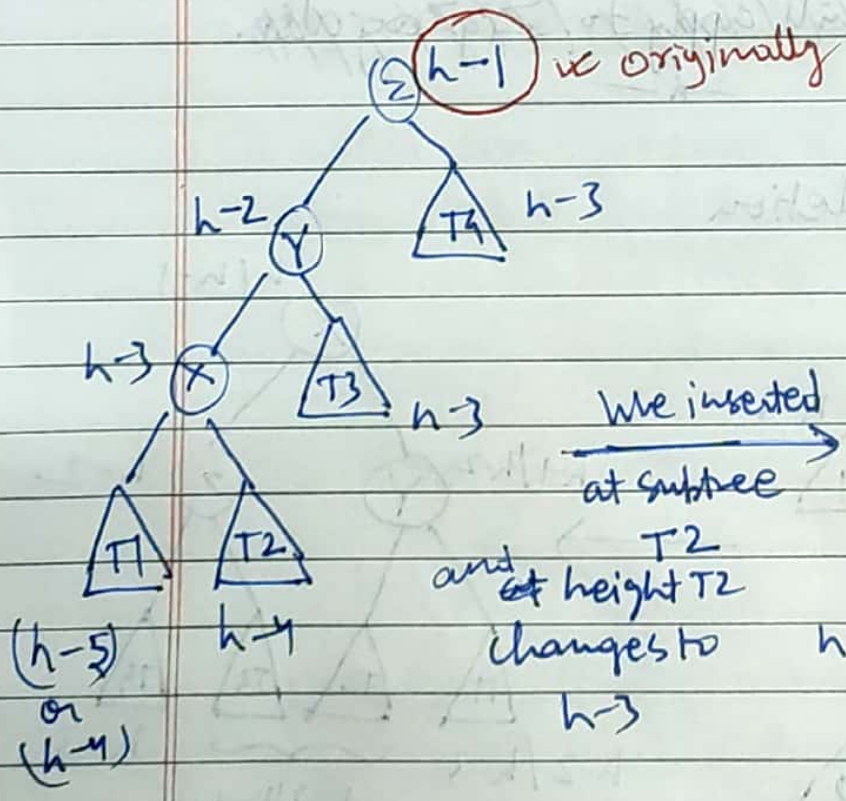
Go to the parent of the inserted node
1.) update the height
2.) check if the AVL property is violated.

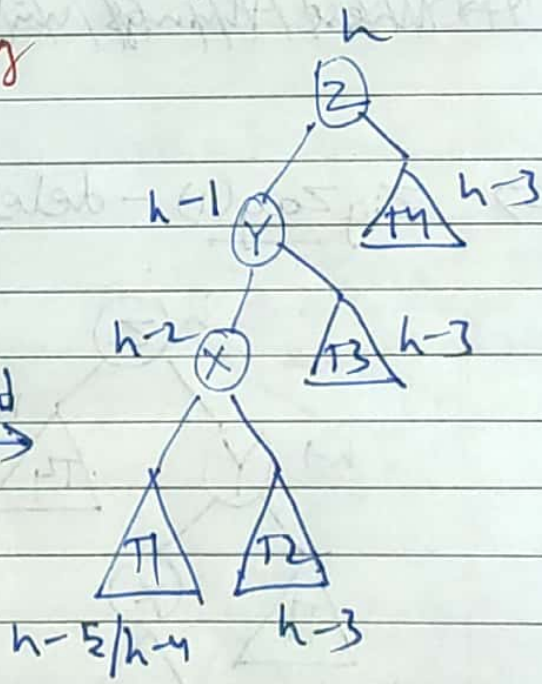
STOP: If height does not change.

→ Z be a node where the AVL property is violated.

Y be the child of Z such that $H(Z) = H(Y)+1$
X " " " " Y " " $H(Y) = H(X)+1$

1st lets take a look at an example:-

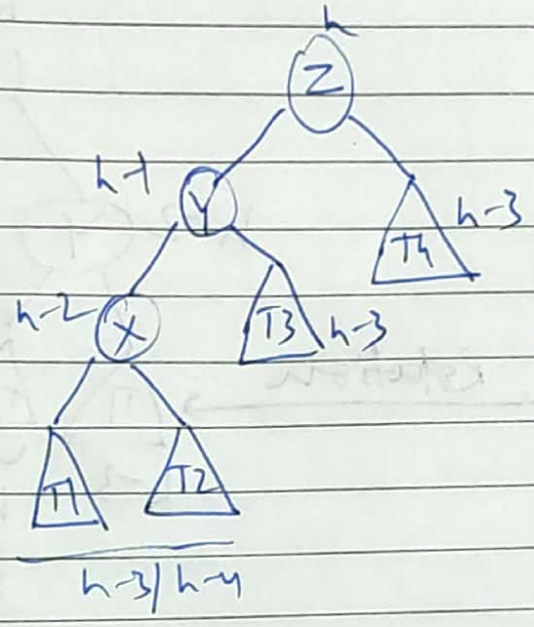




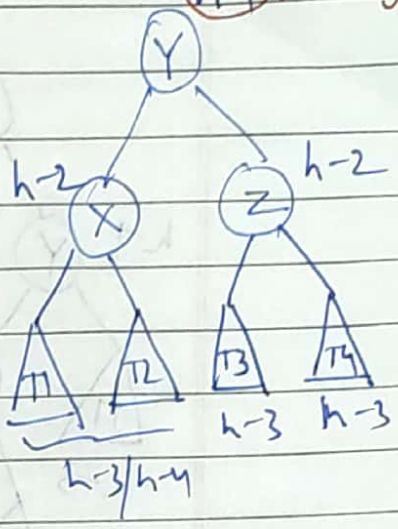
AVL property is violated at Z. This is the kind of example we handle in next pg.

This subtree satisfies AVL property ↑

⇒ ZigZag (0) - Insert



Rotation →



which is why there is no need of another rotation

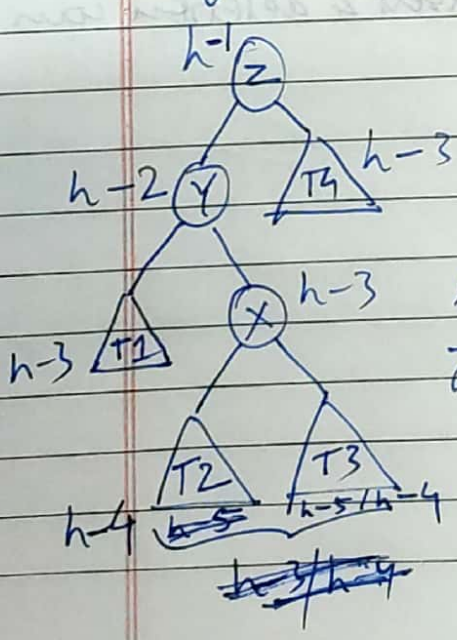
↑ In contrast to deletion

Heights of T1, T2, T3, T4 do not change in rotation.

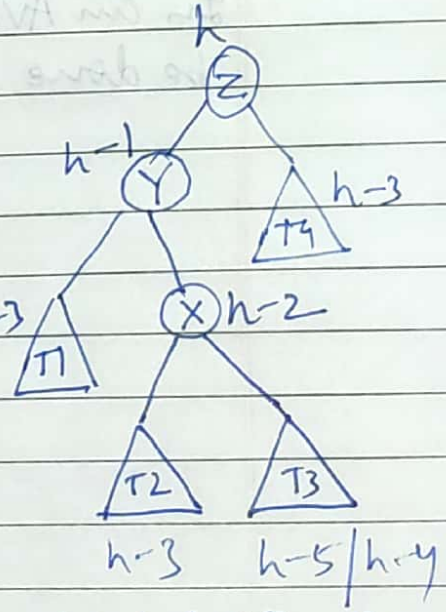
∴ height of the root node changes from h to h-1

⇒ ~~it is necessary~~ no need to check for its parent, whether AVL property is violated or not, b/c originally also the height of the subtree was h+1 (prev. pg)

Similarly lets look at an example for zigzag also.

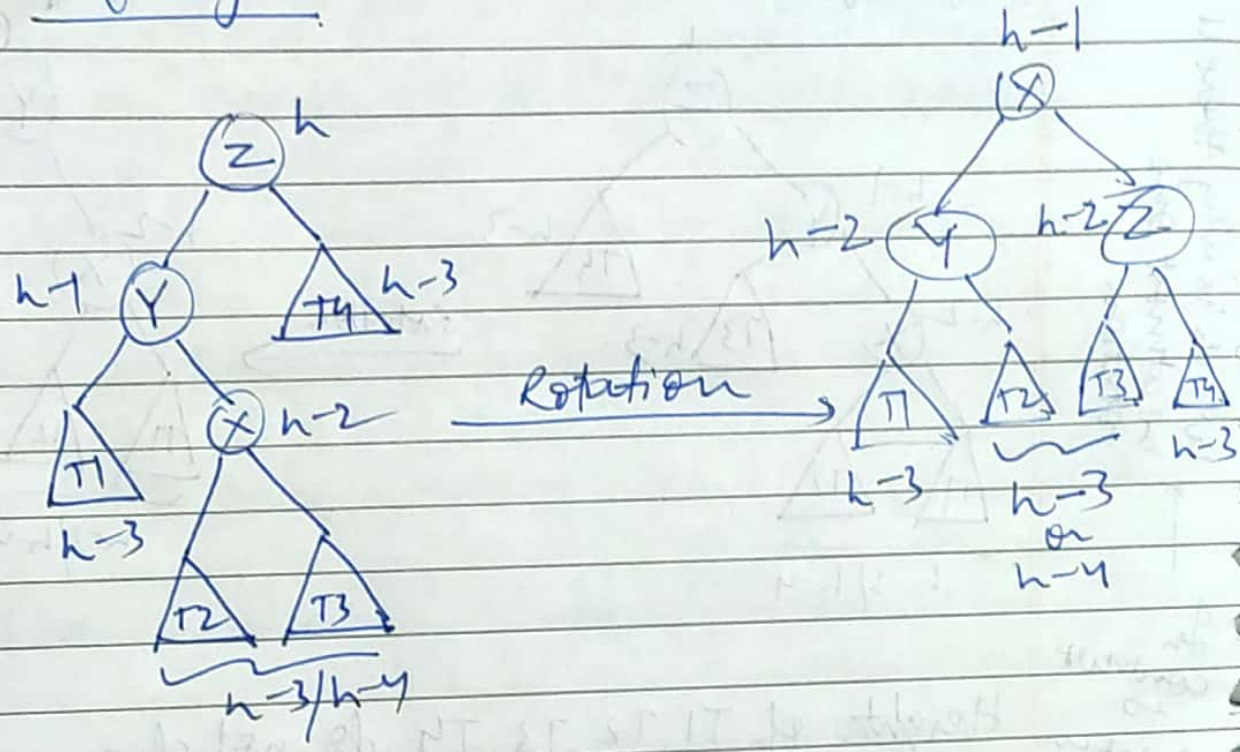


Suppose we inserted at subtree T2 and height changes to h-3 →



AVL property is violated at 2. this is the kind of ex. we handle in the next pg.

⇒ ZigZag (L) - Insert



Summary

Delete: You may have to do O(log n) rotations.

Insert: You'll do atmost one rotation.

In an AVL Tree search, Insert & deletion can be done in O(log n).

We'll implement the functions now. First one being the height fn".

Height(node) → will compute the height of the present node. Since we're going to update the height from the bottom-up fashion, when we come to a node we can safely assume that the height of the children of the node have been computed correctly.

→
```
int Height (struct BST *node)
{
   if (node){
        if (node → left && node → right)
             return ( 1
             + Max(node → left → height,
                   node → right → height) );

        else if (node → left)
             return 1+ node → left → height;

        else if (node → right)
             return 1+ node → right → height;

        else return 0; }

   else return -1;
}
```

2.) AVL(node) → if the AVL property is violated, it returns false, else it returns true.

→
```
bool AVL(struct BST *node)
{
   if(abs(Height(node->left)-Height(node->right)<2)
        return true;

   else
        return false;
}
```

—x—

(*) AVL Tree

Z be a node where the AVL property is violated.

Y be the child of Z such that H(Z) = H(Y)+1
X " " " " Y " " H(Y) = H(X)+1

—x—

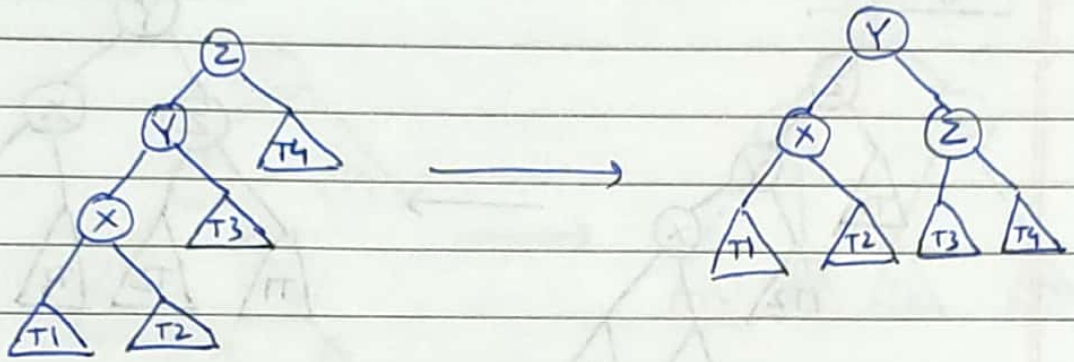
⇒ Define X Δ Y

→
```
void Rotate(struct BST **node, struct BST *Z){
    struct BST *X,*Y,*T1,*T2,*T3,*T4; int c=0;
    if(Height(Z->left)+1==Z->height)  Y=Z->left;
    else{
        Y=Z->right; c++;}
    c=c*2;
    if(Height(Y->left)+1==Y->height)  X=Y->left;
    else  X=Y->right; c++;
```

height of Z, we don't known whether it has been updated or not, that's why we're calling 'Height' fn"

—x—

ZigZig(0)

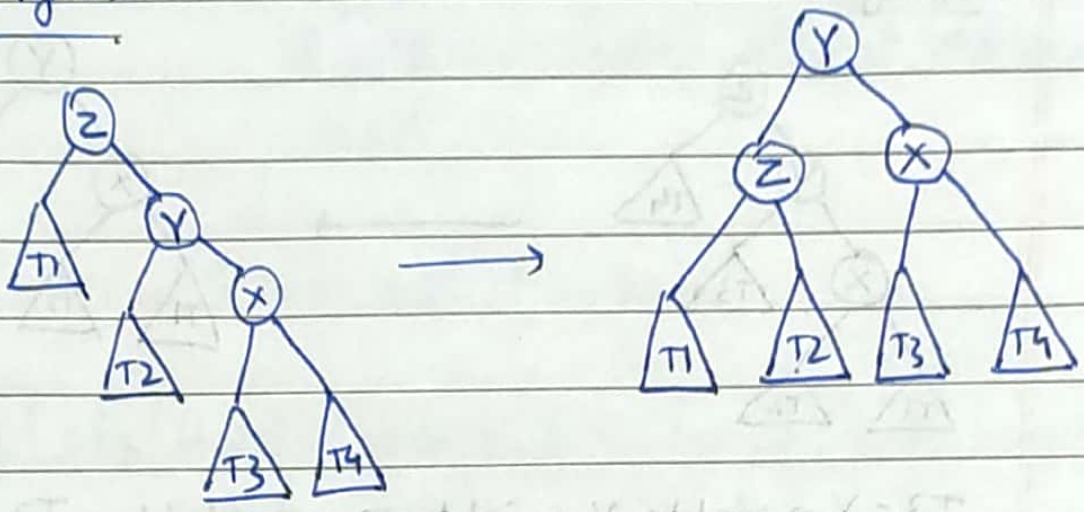


T3 = Y→right; Y→right = Z; Z→left = T3;
Y→p = Z→p; Z→p = Y; T3→p = Z;

Case Code for 0 (Rotate fun cont.)

```
if (c==0){
    if (Z→parent){
        if (Z→parent→left == Z) Z→parent→left = Y;
        else Z→parent→right = Y;}
    else *node = Y;
    T3 = Y→right;
    Y→right = Z; Z→left = T3;
    Y→parent = Z→parent;
    Z→parent = Y;
    if (T3) T3→parent = Z;
    Z→height = Height(Z); X→height = Height(X);
    Y→height = Height(Y);
    }
```

Zig Zig (3)

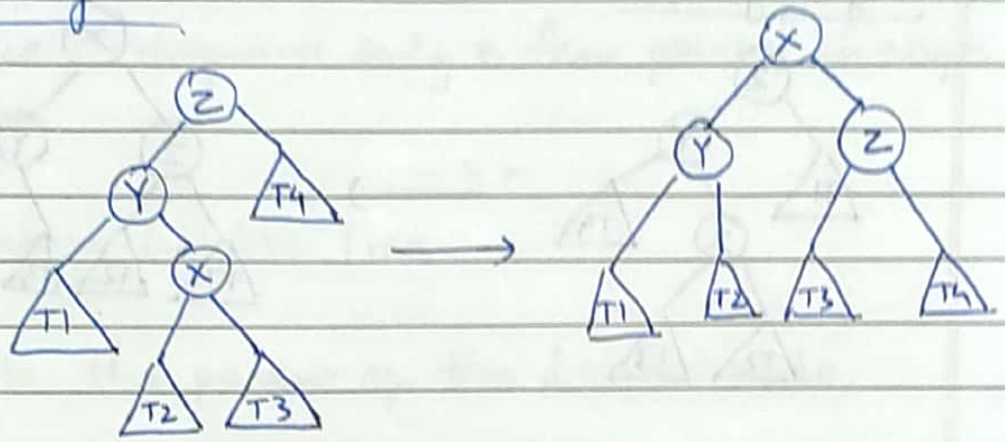


T2 = Y → left ; Y → left = Z; Z → right = T2;
Y → p = Z → p; Z → p = Y; T2 → p = Z;

Code for 3 (Rotate fun cont.)

```
→   else if (c==3){
      if (Z→parent){
        if(Z→parent→left==Z) Z→parent→left=Y;
        else Z→parent→right=Y;}
      else
        *node=Y;
      T2=Y→left; Y→left=Z;
      Z→right=T2; Y→parent=Z→parent;
      Z→parent=Y;
      if(T2) T2→parent=Z;
      Z→height=Height(Z); X→height=Height(X);
      Y→height=Height(Y);
    }
```

Zig Zag (1)

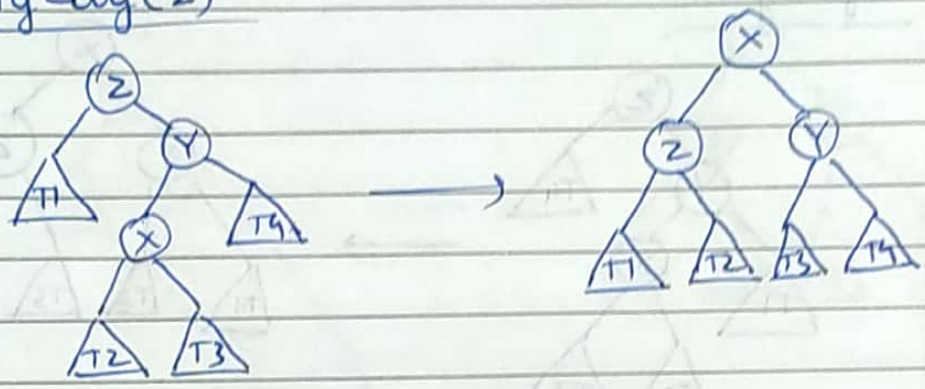


T2 = X → left ; T3 = X → right; Y → right = T2; Z → left = T3;
X → p = Z → p ; Z → p = X; Y → p = X; T2 → p = Y; T3 → p = Z;
X → left = Y ; X → right = Z;

Code for 1 (Rotate fun cont.)

→ . else if (c == 1) {
if (Z → parent) {
if (Z → parent → left == Z) Z → parent → left = X;
else Z → parent → right = X; }
else
*node = X;
T2 = X → left; T3 = X → right;
Y → right = T2; Z → left = T3;
X → parent = Z → parent;
Z → parent = X; Y → parent = X;
if (T2) T2 → parent = Y;
if (T3) T3 → parent = Z;
X → left = Y; X → right = Z;
Z → height = Height(Z); Y → height = Height(Y);
X → height = Height(X);
}

Zig Zag (2)



T2 = X → left; T3 = X → right; Y → left = T3; Z → right = T2;
X → p = Z → p; Z → p = X; Y → p = X; T2 → p = Z; T3 → p = Y;
X → left = Z; X → right = Y;

Code for 2 (Rotate fun^n cont.)

```
else if (c == 2) {
    if (z → parent) {
        if (z → parent → left == z) z → parent → left = x;
        else z → parent → right = x; }
    else
        *node = x;
    T2 = x → left; T3 = x → right;
    Y → left = T3; z → right = T2;
    X → parent = z → parent;
    Z → parent = X; Y → parent = X;
    if (T2) T2 → parent = z;
    if (T3) T3 → parent = Y;
    X → left = Z; X → right = Y;
    Z → height = Height(z); Y → height = Height(Y);
    X → height = Height(x);
}
```

Rotation fun^n ends

Hence, rotation is a constant time operation, i.e. O(1) since it requires only a few pointer changes in any case.

—x—

Deletion in AVL Tree

Go to the parent of the deleted node.

1. Check if the AVL property is violated. If it is then call the rotate fn
2. If AVL property is satisfied then see if the height needs to be updated.

If the AVL property is not violated AND the height is not changing then your algorithm stops.

This is what you do after you delete a node.

—x—

Notice that there were three cases when you did the deletion of a node. We'll not discuss case 2 b/c it's only replacement of a node & then you'll recursively delete the inorder predecessor. So we need to consider only the case0 & case1. In either of these 2 cases, we'll cut a node & replace the child of that node with the parent. So, when we do this operation, the height of the parent can change.

In the following code, temp is the parent pointer of a node that you deleted.

** WHOLE CODES ARE ALREADY ON LMS (NOTES FOLDER)

AVL with parent pointer.

Date __/__/__
Page

→ struct BST *temp = p; bool flag;
if (p) flag = true;
else flag = false;
int h;

while (flag) {
condition 1 → if (!AVL(temp)) {
Rotate (&*root, temp);
temp = temp → parent → parent;
reaching the root node ← if (!temp) flag = false; }
condition 2 → else {
h = Height (temp);
if (h < temp → height) {
temp → height = h;
if (temp → parent)
temp = temp → parent;
else
flag = false; }
}
else
flag = false; // both conditions are false.
}
// end while

→ BST without parent pointer

→ just a part of whole code

→
```
struct BST* Delete(struct BST *node, long long int X)
{
    if(node->key > X){
        node->left = Delete(node->left, X);
        node->height = Height(node);
        if(!AVL(node)) node = Rotate(node);
        return node; }

    else if(node->key < X){
        node->right = Delete(node->right, X);
        node->height = Height(node);
        if(!AVL(node)) node = Rotate(node);
        return node; }
```

In above code, we're not using the fact that, if the AVL property is not violated and height also doesn't change, then we want to stop, that's what the iterative algo. does. But in above case we don't know how to stop it. So it will go from the node where it's deleted, all the way upto ~~to~~ root; check whether the height is diff or the AVL property is violated.

However, it's ok to go all the way upto the root, b/c even then the no. of ~~com~~ comparisons will be of the order $\log n$.

(*) Insert in AVL Tree.

Go to the parent of the inserted node

1.) update the height.
2.) check if the AVL property is violated.

STOP: If height does not change.

—x—

The code written below is, after you insert a node.
'temp' is the parent of a node ~~where~~ you inserted.

→ BST with parent pointer

```
→ flag = true; int h
while (flag) {
        h = Height (temp);
        if (h > temp → height) {
            temp → height = h;
            if (!AVL (temp)) {
                Rotate (&*node, temp);
                flag = false; }
            else if (temp → parent)
                temp = temp → parent
            else
                flag = false; }

        else
            flag = false;
            }
```

→ BST without parent pointer

```
struct BST *Insert(struct BST *node, long long int key)
{
    if(key < node->key)
        node->left = Insert(node->left, key);
    else if(key > node->key)
        node->right = Insert(node->right, key);

    node->height = Height(node);
    if(!AVL(node))
        node = Rotate(node);
    return node;
}
```

—×—

(*) Some more operations other than search, delete and insert.

⇒ Rank(x) = 1 + (no. of numbers in the collection > x)

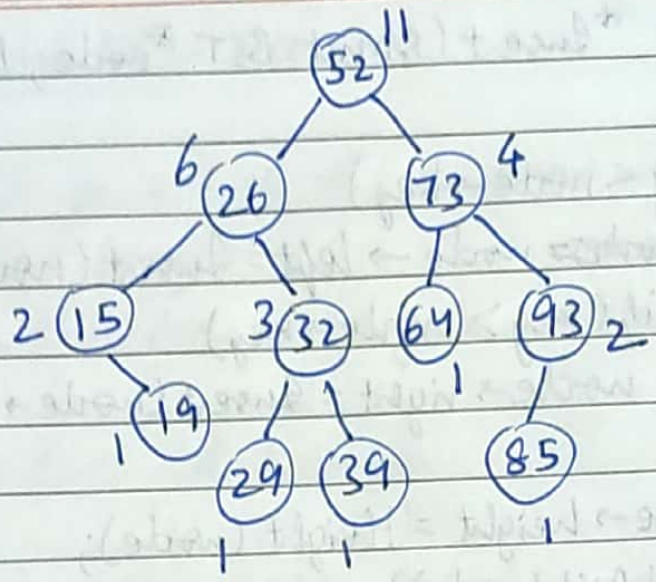
In order to implement this in the structure of the BST node which is 'count'.

→
```
struct BST{
    long long int key;
    int height;
    long long int count;
    struct BST *left, *right, *parent;
};
```

—×—

Def:- Count will store the no. of nodes rooted at that node, in that subtree.

Ex:-



Count of each node is written beside it.

→ while inserting as you travel along the tree, increment the count of each node by 1.

Ex:- we need to insert 40.

we'll start at 52, ~~↑e count to 12~~

52 → 26 → 32 → 39 & the insert

↑e count to 12 | ↑e count to 7 | ↑e count to 4 | ↑e count to 2

Similarly, while deleting ↓e the count by 1.

Ex:- we need to delete 39.

52 → 26 → 32 & delete 39

↓e count to 10 | ↓e count to 5 | ↓e count to 2

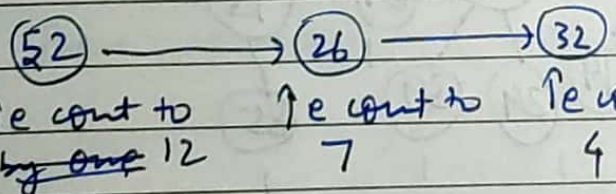
→
```
long long Count (struct BST *node){
    if (node){
        if (node->left && node->right)
            return (1 + node->left->count
                    + node->right->count);

        else if (node->left)
            return 1 + node->left->count;

        else if (node->right)
            return 1 + node->right->count;

        else return 1;}

    else return 0;}
```

—x—

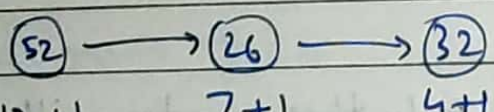
Point to be noted is that, this count will solve the problem of adding the same no. ~~###~~ multiple times in the tree. How does it solve, we can see from ex:- given below:-

Suppose in the tree given in prev. pg we want to add two more 32's.

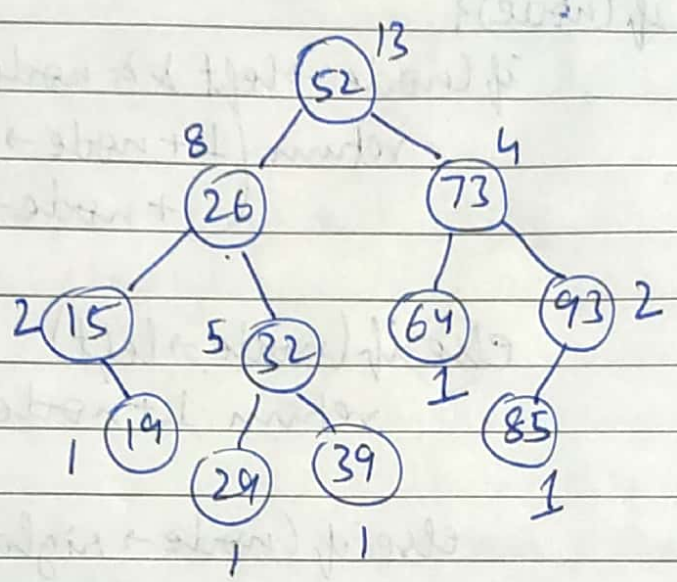
adding the 1st 32



adding the 2nd 32



The final tree will look like



The freq. of a no. can be found out as follows:-

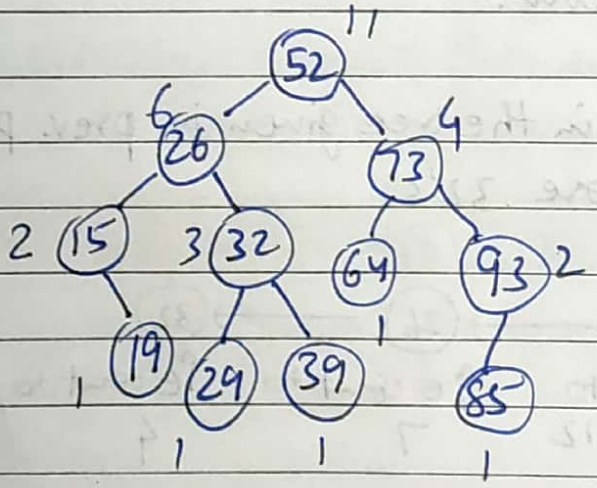
(★★) freq. of key in the node = (Count of node) − [(Count of left child) + (Count of right child)]

i.e. freq. of 32 = 5 − (1 + 1) = 3

—×—

Lets see how we find rank of a node through ex:-

Ex:-



1.) Find rank of 19.

Ans → Keep in mind that only those nodes will be significant whose key values are greater than 19.

Let 'r' store the rank

We'll start with the root node (52)

$52 > 19$

$\Rightarrow r = (\underbrace{\text{no. of numbers} > 52 \text{ in the current tree}}) + (1)$ ← 52 itself

$= (\text{count of right child of } 52) + 1$

$= 4 + 1 = 5$

Now we go to 26

$26 > 19$

$\Rightarrow r = r + (\text{no. of numbers greater than 26 in current subtree}) + 1$

$= 5 + (\text{Count of right child of } 26) + 1$

$\Rightarrow r = 5 + 3 + 1 = 9$

Now we go to 15; $15 < 19$, so we do nothing.

Then we reach 19 $\Rightarrow r = 9 + 1 = 10$

$\therefore$ **rank of 19 = 10**

2.) Find rank of 32

52 → 26 → 32

$r = 4 + 1 = 5$ $r = 5$ $r = 5 + 1 + 1 = 7$

Taking the right subtree of 32 into account

n = no. of nodes in the tree.

Date / /
Page

⇒ Rank(x) → O(logn)

→
```
long long int Rank(struct BST *node, long long int x)
{
    long long int rank = 1;
    while(node)
    {
        if(x == node->key){
            if(node->right) rank += node->right->cont;
            return rank;}

        else if(x < node->key){
            rank++;
            if(node->right) rank += node->right->cont;
            node = node->left;}

        else
            node = node->right;
    }
    return rank;
}
```

→ x ←

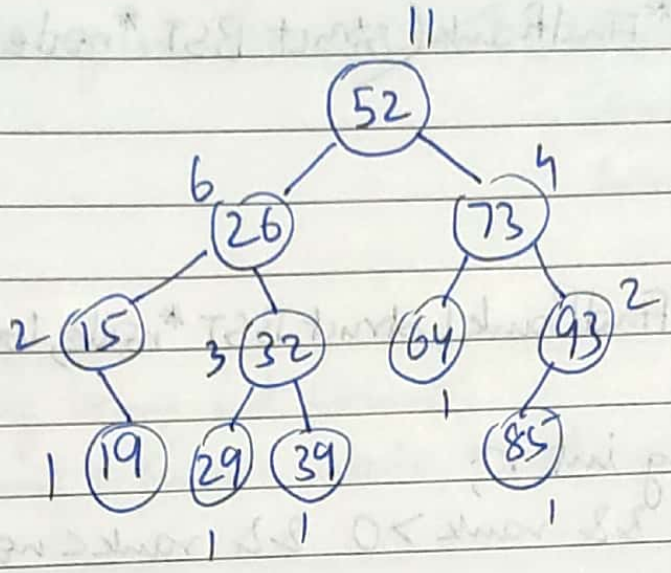
(*) Find Rank(rank) → O(logn)

Given rank, find a no. X, such that Rank(x) = rank.

Q. How does Find Rank work?

Sol" Start from the root node, if the rank is bigger than root, move to left. If it is smaller, move to right. Update the rank whenever you move left

Ex:-



Let rank = 6 then find the no.

we'll start with 52

Rank(52) = 5 < 6

So we move to left, to 26

rank = rank − 5 = 6 − 5 = 1

Rank(26) in current subtree = 4 > 1

So, we move to right, to 32

Rank(32) in current subtree = 2 > 1

So, we move to right, to 39

Rank (39) in current subtree = 1 = 1

we stop.

→ struct BST *FindRank(struct BST *node, long long int rank)
{

}

→
```
struct BST *FindRank(struct BST *node, long long int rank)
{
    long long int r;
    if(node && rank>0 && rank<node->count+1)
    {
        while(node)
        {
            if(node->right) r=node->right->count+1;
            else r=1;
            if(r==rank) return node;
            else if(rank>r)
            {
                node=node->left;
                rank=rank-r;
            }
            else node=node->right;
        }
    }
    return NULL;
}
```

→x—

(*) Rangecount(l, r) → O(log n)

~~no.~~ This will give the no. of x. such that

$$l \leq x \leq r$$

For Ex in the tree in prev pg

if l = 52 r = 93 Rangecount returns 5.

→ long long int Rangecount (struct BST * node,
long long int l,
long long int r)

```
{
    long long int count;
    count = Rank(node, l) - Rank(node, r);

    if (Search(node, l)) count++;
    return count;
}
```

—x—

(#) Other operations (All operations mentioned below are O(log n))

1.) PrefixSum(X) = find the sum of all the numbers smaller than or equal to X.

write the code (H.W)

For this you have to include one more item to the structure of your node which is 'sum'

where node → sum = node → left → sum
+ node → right → sum
+ node → key.

This is analogous to count

node → count = node → left → count
+ node → right → count
+ 1

2.) RangeSum(X,Y) = sum of no. b/w X & Y, including X & Y.

Analogous to RangeCount

RangeSum(X,Y) = PrefixSum(Y) − PrefixSum(X)

3.) Average(X,Y) = $\frac{\text{RangeSum}(X,Y)}{\text{RangeCount}(X,Y)}$

4.) Morethanaverage(X,Y) = the no. of numbers b/w X & Y, bigger than Average(X,Y)

= Rank(Y) − Rank(Ang(X,Y))