Heap

Max heap

if each node N of h has the property that the value at N is ≥ the value at each of the children. Consequently, the value at N is ≥ the value at any descendant of N

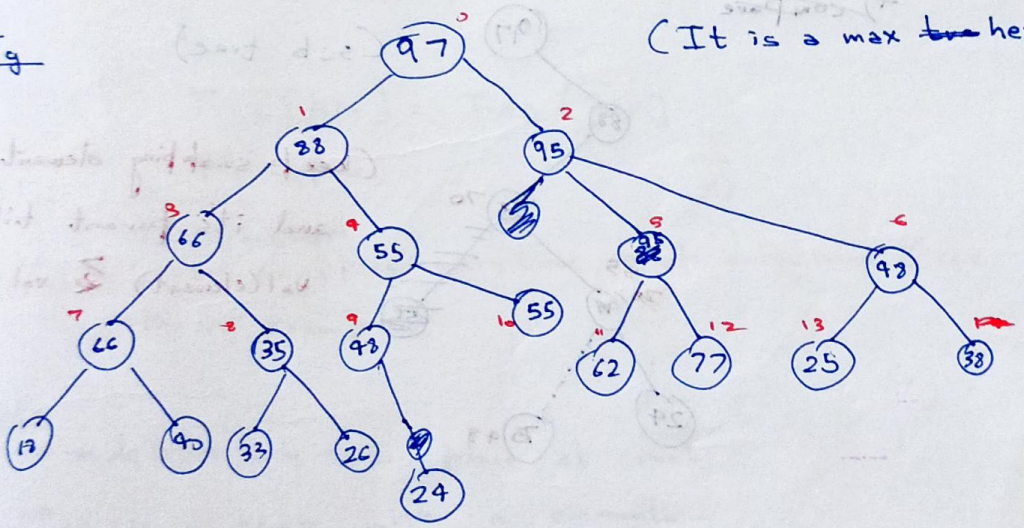
→ No relation for same gen nodes

Heap is a complete Binary Tree

Minheap

→ similar to ~~min~~ max heap, but ≤

Eg



(It is a max ~~tree~~ heap)

→ we can use linked list or sequential

→ but as it's complete Binary tree, using sequential is better.

Tree

97	88	95	66	55	95	98	66	36	--	—	—

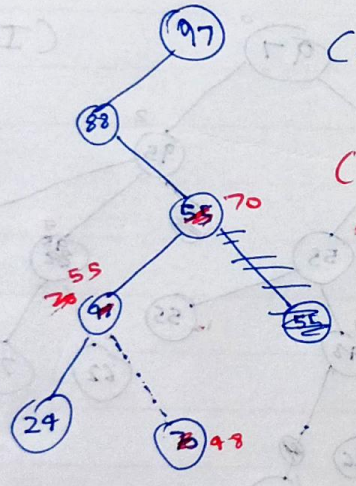
Inserting a node in heap H

1) adjoin the new element at the end of H, so that H still remains a complete Binary Tree, but not necessarily a heap.

2) Let the new element vise to it's proper place, so that H finally becomes a Heap.

Eg: suppose we want to insert 70 in prev Tree.

~~1) compare~~



(sub tree)

(keep swapping element and it's parent till val(element) ≥ val(parent)

Procedure

```
insert_heap ( int tree[], int n, int ITEM )
{
    n = n+1;
  int ptr = n;
    while (ptr ≠ 1)   // root is at 1
    {
    int par = L ptr/2 ]   // floor (ptr/2)
      if ( item < Tree[par])
      {     Tree[ptr] = item;
            return;
      }
      Tree[ptr] = Tree[par]
      ptr = par;
    }
Tree[1] = ITEM
```

// rather than swapping every time, swap only once (see with an example).

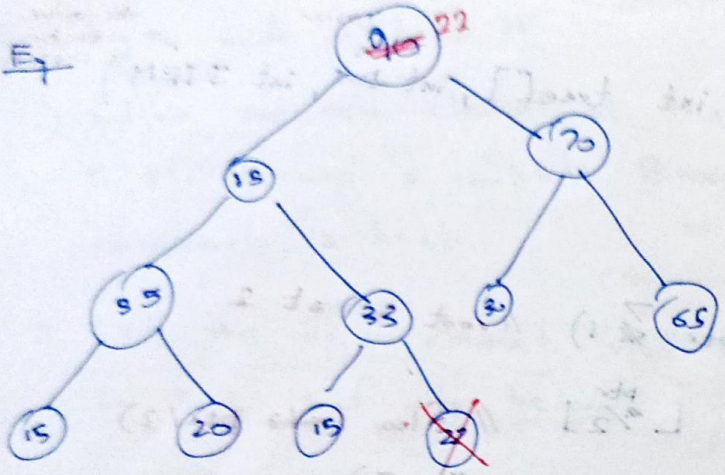
Delete

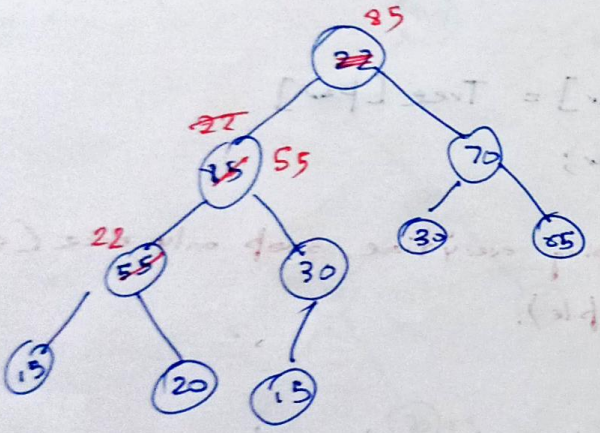
→ deletion only takes place at root

→ H is a heap with n elements

1) item = value at root
2) Replace deleted node. R with the last last node L of H, so that H is still complete Binary Tree, but not necessarily a heap.
3) Reheap: Let L sink down to it's appropriate position so that H becomes heap.

Ex





check childs, swap element with max value of child till val of childs are less than elem

Delete

Procedure

```
void delHeap ( int Tree[], int n, int *item)
{
    *item = Tree[1]
    last = Tree[n]
    n = n-1;
    ptr = 1 ; left = 2 , right = 3;
    while (right <= n)
    {
        if (last >= Tree[left] && last >= Tree[right])
        {
            tree[ptr] = last;
            return;
        }
        if (Tree[right] <= Tree[left])
        {
            Tree[ptr] = Tree[left];
            ptr = left;
        }
        else {
            Tree[ptr] = Tree[right];
            ptr = right;
        }
        left = 2*ptr ;
        right = 2*ptr+1 ;
    } //end of while.
} //end of fn.
    if (left == n) {
        if (last < tree[left]) {
            Tree[ptr] = last; ptr = left;
        }
        tree[ptr] = last;
} //end of fn.
```

Heap sort

→ Due to structure of Heap, we can use heap to sort

→ Procedure

→ add all value to heap

→ delete all value from heap, output will be sorted

Complexity : $n(\log n)$

- n values of array
- per value time complexity (cause binary, $\log_2 n$)

Priority Queue using heap

→ if we add priorityies, we see that highest priority element will be processed first.

Balanced Binary Search Tree

→ In normal BST, complexity of operation depth of tree

→ if tree becomes skew, complexity bad.

→ We can use Balanced BST to solve this

[we ~~called~~ considering AVL Tree & Red-Black Tree]

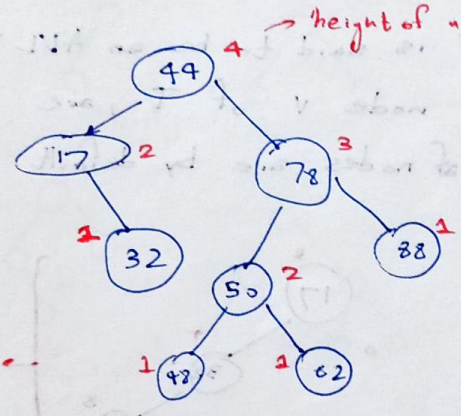
AVL Tree

A → Adilson

V → Velsky

L → Landis

Eg:



→ consider height of a node by the highet of subtree with the node as root → unlike height of tree which is counted opposite (bottom to upward counting)

→ height of leaf = 1

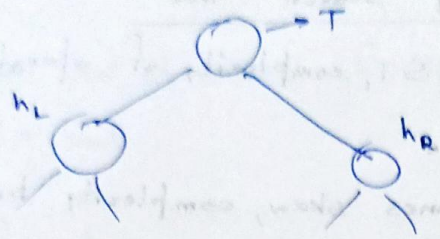
→ heigh of node = max (height of lsubtree, heigh rgnt) + 1

→ A BST is AVL Tree if the tree is height balanced.

Definition

An internal node of the Tree T, is said to be height balanced iff the height of the children node differs by at most 1

Eg:



T is said to be balanced, iff

$$0 \leq |h_L - h_R| \leq 1$$

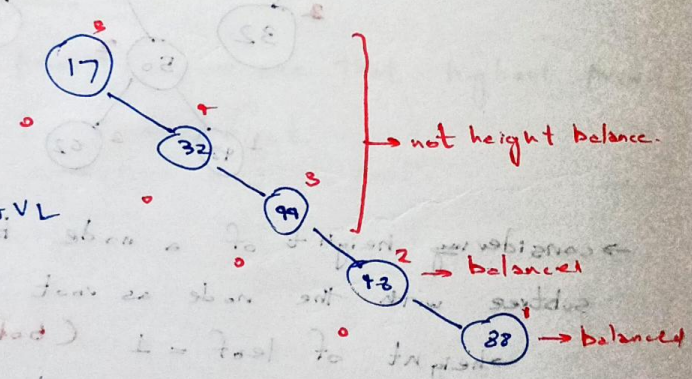
or $|h_L - h_R| = \{0, 1\}$

Definition

→ A BST is said to be an AVL Tree iff every internal node v of T, are height balanced.

→ Leaf nodes are by default balanced.

Eg:



→ This is not AVL

Why AVL ?

→ Balance Tree complexity is approx. $O(\log n)$ rather than $O(n)$, for a skew tree

Theorem

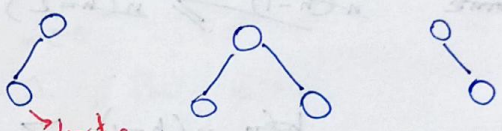
→ The height of an AVL tree • contain n keys is ~~$\log_2 n$~~ $O(\log_2 n)$

Proof: Let us consider all possible AVL trees of height h. Let us also take one AVL Tree, out of many tree, which has smallest no. of nodes (n)

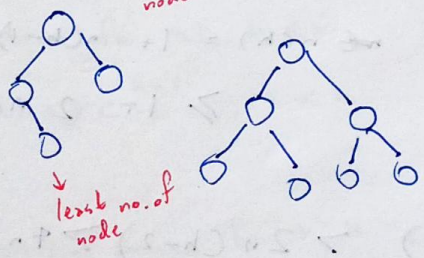
if, $n = c^h \rightarrow h = \log_c n$

where c is some constant

Eg h = 2



h = 3



Let us define that quantity to be

$n(h)$: min. no. of nodes of an AVL Tree with height h

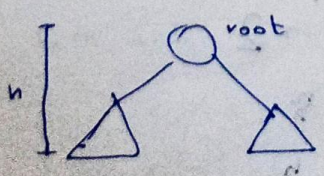
→ Given an AVL Tree of height h

$n(1) = 1$

$n(2) = 2$ (not 3)

$n(3) = 4$

→ for an AVL Tree of height h, at least one of the subtree of root has height h-1, and other can have (h-1)/(h-2)



→ for making min. no of nodes, we take the case where one subtree has height $h-1$ and other has height $h-2$.

$$\therefore n(h) = \begin{cases} 1 & h=1 \\ 2 & h=2 \\ 1 + n(h-1) + n(h-2) & \text{otherwise} \end{cases}$$

→ now $n(h-1) \geq n(h-2)$

also $n(h) = 1 + n(h-1) + n(h-2)$

$\geq 1 + 2\,n(h-2) > 2n(h-2)$

$\therefore n(h) > 2n(h-2) > 4n(h-4) \ldots\ldots$

$\therefore n(h) > 2n(h-2) > 4n(h-4) \ldots\ldots > 2^i n(h-2i)$

→ we know $n(2) = 2$

$\therefore$ if we assume $n(h-2i) = 2$

it implies $h-2i = 2$

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

$\therefore n(h) > 2^{\frac{h}{2}-1} \times 2$

$\therefore n(h) > 2^{\frac{h}{2}}$

$n(h) > (\sqrt{2})^h$

→ If the minimum no. of nodes is m, then

$$m > 2^{\frac{h}{2}}$$

or $h < 2 \log_2 m$

∴ AVL Tree of m nodes have ~~height~~ height of $O(\log_2 m)$

↓

order of $\log_2 m$

Another method

~~we ass~~ Proving by Induction that $n(k) \geq c^{k-2}$

Base ~~k~~ ~~n=1~~

$n=1$, $n(h) = 1$, for $c > 1$

Induction hypothesis

→ claim is true for all $h < k$

$$n(h) \geq c^{n-1}, \quad h < k$$

Induction step

→ we have to prove for $n=k$, i.e. $n(k) \geq c^{k-1}$

The earlier recurrence was

$$n(k) = n(k-1) + n(k-2) + 1$$
$$n(k) \geq c^{k-2} + k\, c^{k-3} + 1$$

ignoring 1

$$n(k) \geq c^{k-2} + c^{k-3}$$

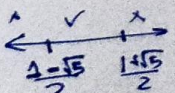
we can prove $n(k) > c^{k-1}$

if $c^{k-2} + c^{k-3} \geq c^{k-1}$

dividing both side by c^{k-3}

$$c + 1 \geq c^2$$

$$c^2 - c - 1 \geq 0$$

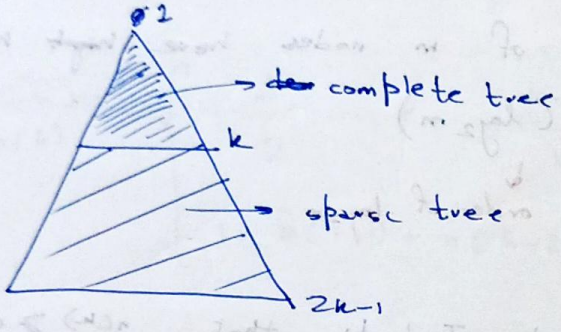


$c = \left[\frac{1-\sqrt{5}}{2}, \frac{1+\sqrt{5}}{2}\right]$, is we going for ratio

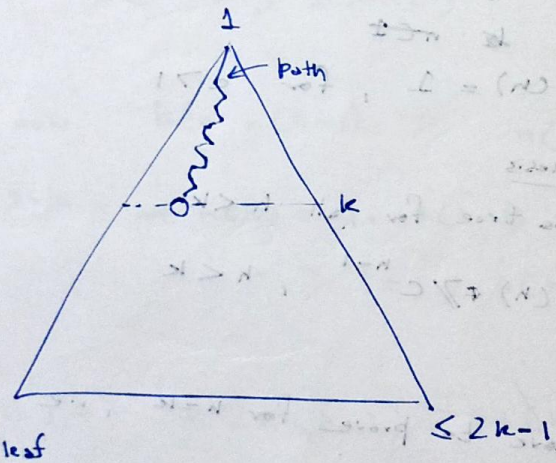
$c = \frac{1+\sqrt{5}}{2}$ → golden

Proposition

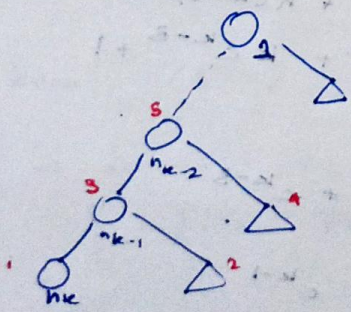
- If the level of the closest leaf in an AVL Tree is k, (assuming root at level 1). Then height of the AVL Tree is at most $2k-1$



Proof



→ Let the leaf node at level k be n_k



→ n_k's height is 1

→ hence height of right subtree of n_{k-1} is 0/1/2

~~Gate~~, we take it to be 2 as we need largest tree.

→ similarly degree of right subtree should be 4, making degree n_{k-2} to be 5

→ ~~hence~~ similarly $h_{k-3} = 7$

$h_{k-4} = 9$

$\therefore h_{k-i} = 2i + 1$

now if $k - i = 1$ or $i = k - 1$

$h_1 = 2k - 2 + 1$

$= 2k - 1$

$\therefore$ max possible height $\Rightarrow 2k-1$

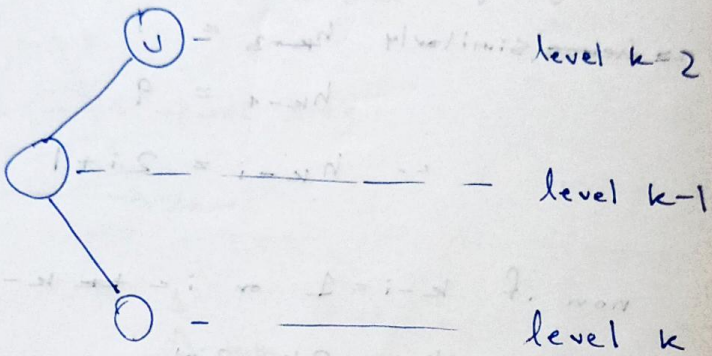
$\therefore$ height $\leq 2k-1$, when smallest leaf node has height k

Proposition

If the closelest leaf is at level k, then all the node at level 1 through k-2 have 2 children.

Proof by contradiction

let



→ suppose the exist a non-leaf node u at level k-2 with ~~dait~~ children < 2 i.e, it has 1 child as it not leaf.

→ now the left subtree of u have height 2, but right-subtree have height 0, hence, the tree fails to be AVL, which is a ~~contr~~ contradiction.

∴ The no. of nodes in an AVL Tree with smallest leaf node at level k, ~~has~~ has the property

$$2^{k-1} \leq n \leq 2^{k-1}$$

where n is the no. of nodes.

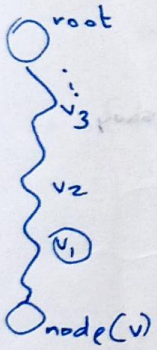
Insertion in AVL Tree

→ adding a node is similar to BST insert, but that can add height to some node, and can make Tree unbalanced.

→ If tree becomes unbalance some nodes have to be shifted.

→ Height Imbalance: $|h_l - h_r| > 1$

→ we find first imbalance node in the path between root and inserted node, if we configure that node, all othe nodes can be repaired.



→ Hence we need to find first ancestor node of v (let it be v_1)

→ then we rearrange subtree with root v_1, to make it AVL

→ This can make every ~~node~~ ancestor of node balanced.

→ we also keep parent pointer in a node for easy traversal.

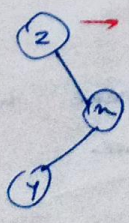
Convention

→ Let the first ancestor node to be unbalanced be z

→ n is a node in a path from v to root, which is grandchild of z (or z is grand parent of n)

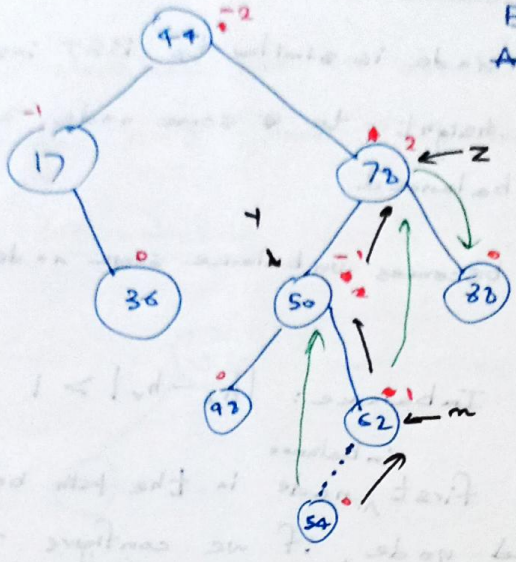
→ y : y is parent of n and z is parent of y

Eg:



→ first unbalanced node

Eg



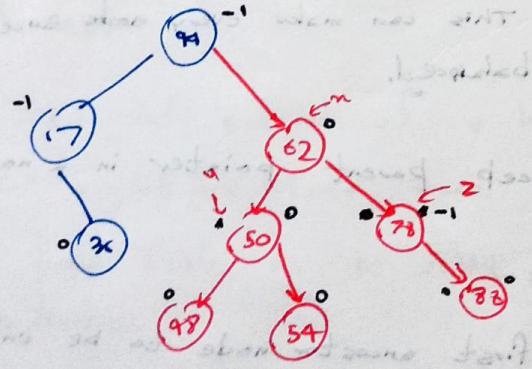
BST ✓

AVL ✓

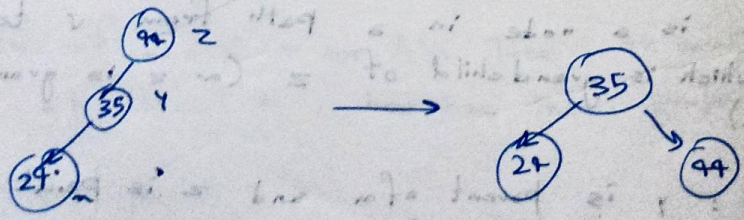
→ after inserting 54, tree is not AVL

→ as we go reverse tree, parent is necessary

→ After rearrangement, we get



→ Eg:

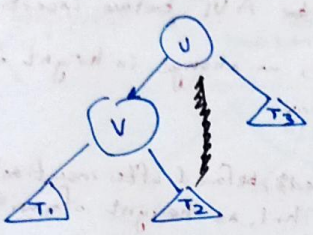


→ There can't be a case where n is NULL, as that case can't have unbalance

→ The arrangement we have done by a process called "Rotation".

Rotation

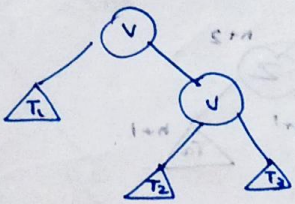
→ Locally Reorganization of tree



→ By BST, (all values in T_1) < v < (all values in T_2) < u. . .

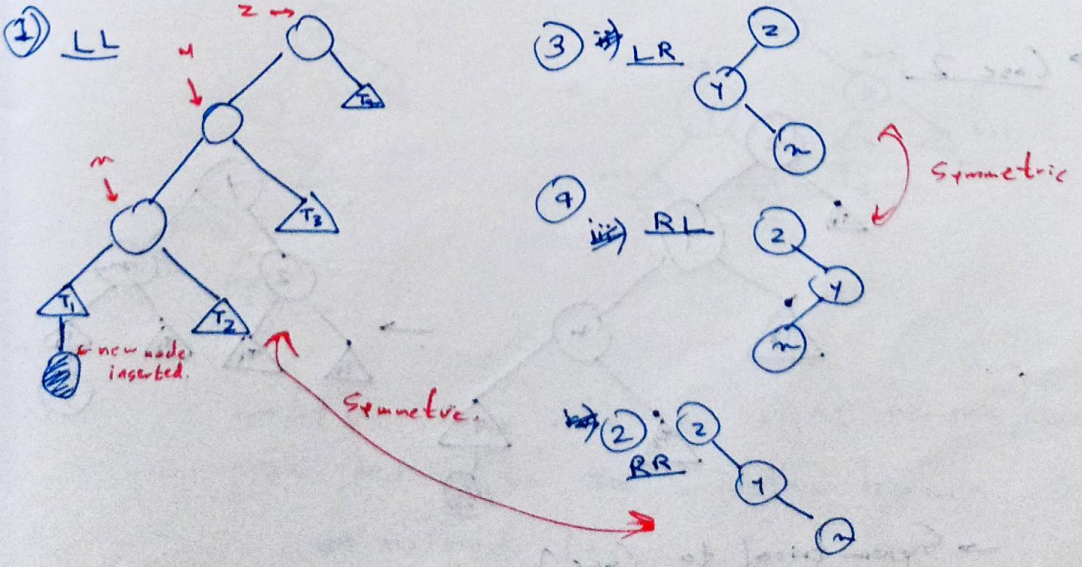
or keys(T_1) < v < keys(T_2) < u < keys(T_3)

→ after rotation, we get

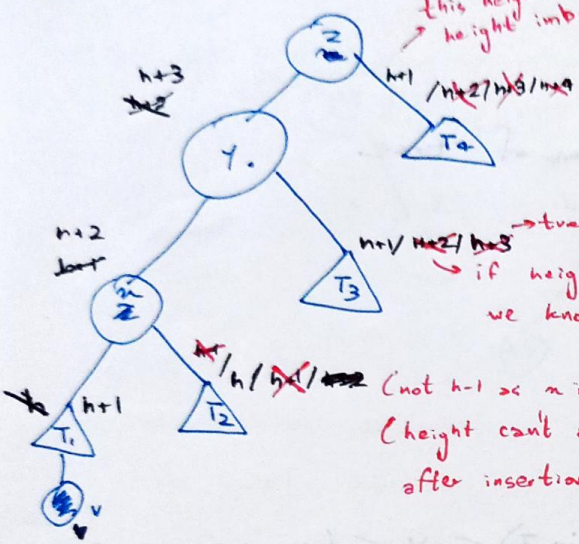


here keys(T_1) < v < keys(T_2) < u < keys(T_3)

Use of rotation in AVL Tree insertion : 4 cases



Case 1



this height being height imbalance → suppose height $(T_1) = h$

→ z is imbalanced because height of x & y is incresed.

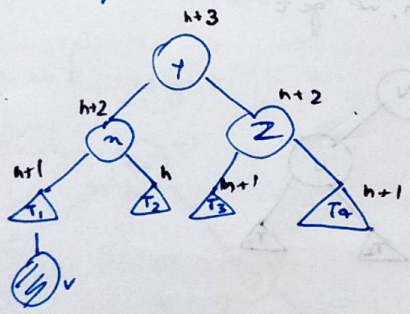
→ true ~~can't be~~ is AVL before insertion

→ if height h+2, no change in height of y, which we know is true.

(not h-1 as x is balanced before & after insertion)

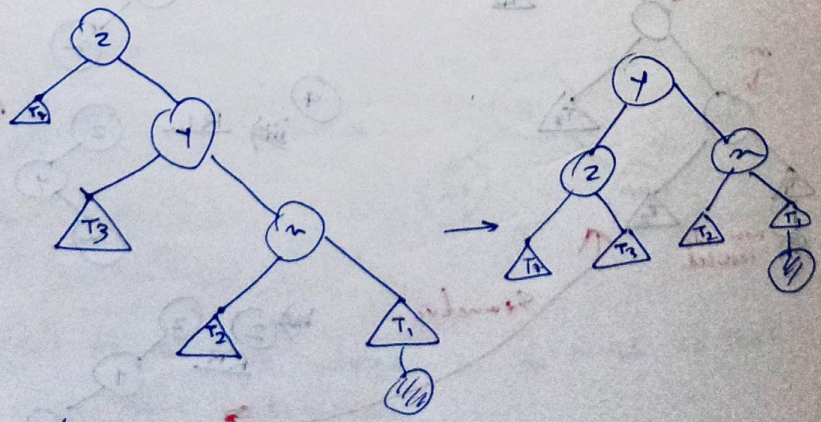
(height can't also be h+1, as height of x will not increase after insertion)

→ After rotation, we get



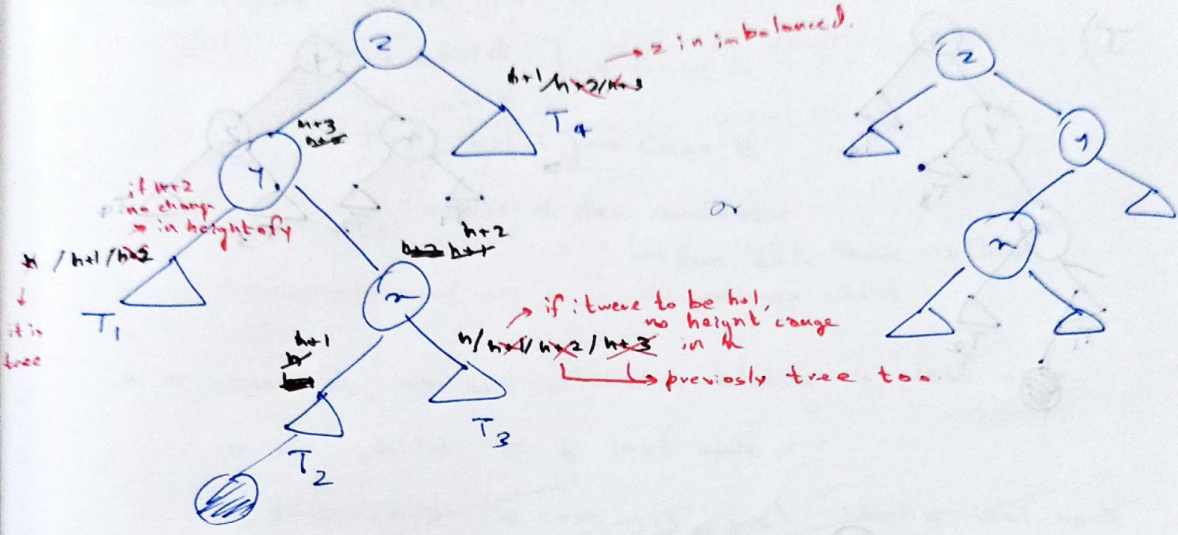
→ hence the tree is now balanced.

→ Case 2:



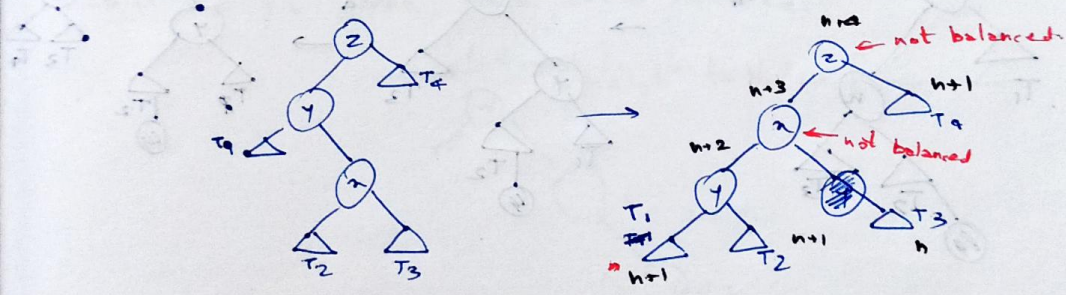
→ Symmetrical to Case 1

Case 3 or Case 4

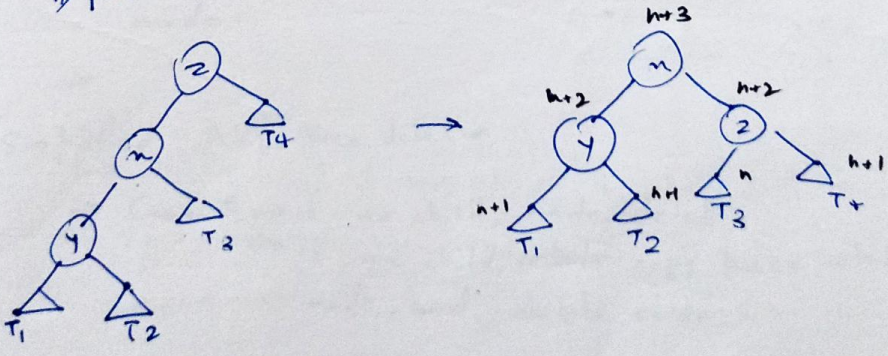


→ We do double rotation.

i) rotation around x & y.



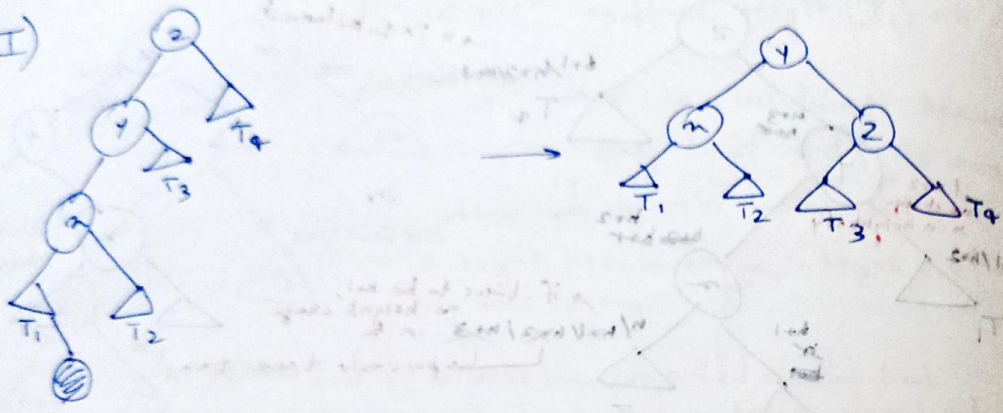
ii) perform rotation around x & z



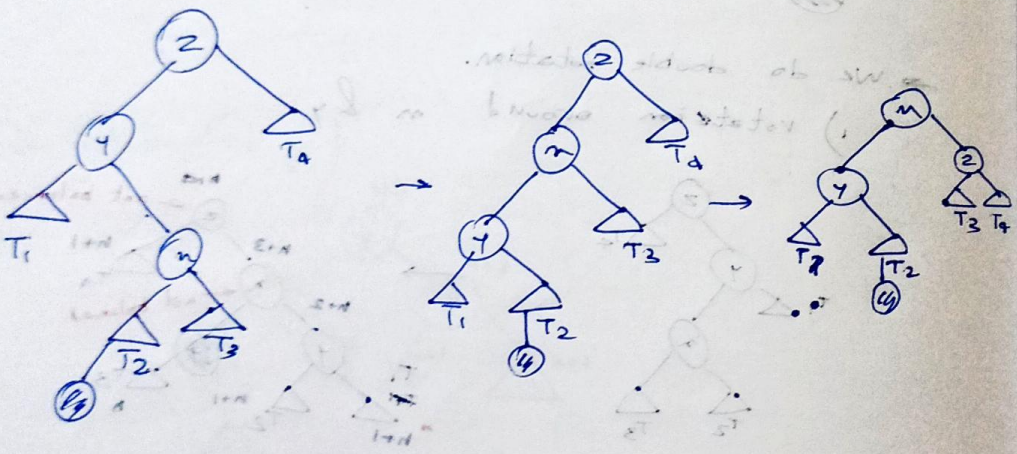
→ After ~~change~~ rotations, height remains same as before, so the ancestors becomes ~~bal~~ balaced too

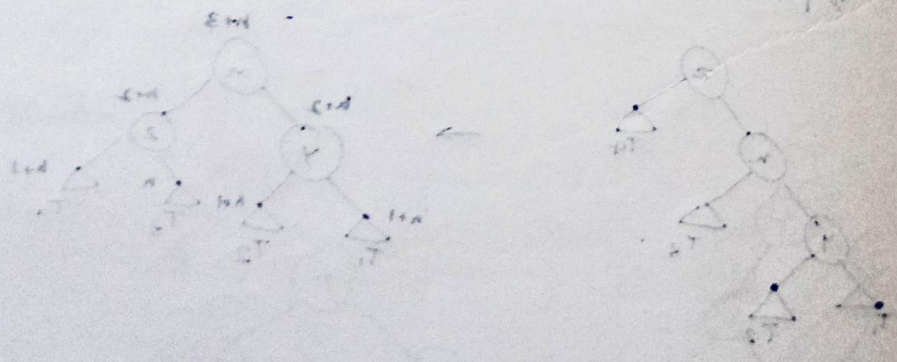
Summary

I)


z
y
x
T1
T2
T3
T4
y
x
z
T1
T2
T3
T4

II)


z
y
T1
x
T2
T3
T4
z
x
T4
y
T3
T1
T2
x
y
z
T1
T2
T3
T4



Deletion of Node in AVL Tree

→ Three case for BST Tree deletion

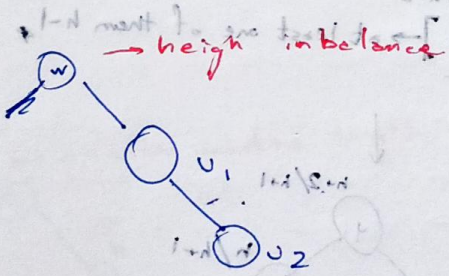
i) No child] → Case A
ii) One child]
iii) Two child] → Case B
↳ in order successor
↳ can only have rchild or no child

→ in case A, we are ultimately deleting a leaf node.

→ no child is a leaf node.

→ ~~one child is case, is just like a leaf node~~
~~data copied then leaf deleted.~~

→ in AVL Tree, some extra nuisance.



→ ie in one child case, child will always be leaf node.

→

→ Simplified AVL Tree deletion

→ Case A: → if no child, node deletion
→ if one child, ~~delete~~ copy paste child to node, and delete node.

→ Case B: → find in-order successor of node (in order successor have only one rchild)
→ copy paste successor to node, and delete in order successor (ultimately leaf node)

→ i.e, per deletion, one leaf node gets deleted.

→ Convention

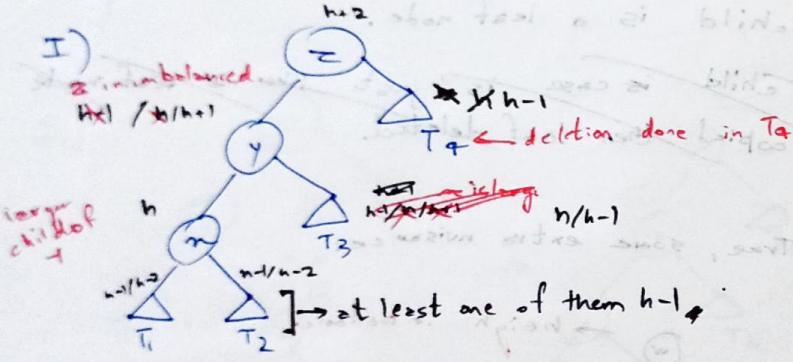
→ z : first ancestor to have height imbalance.

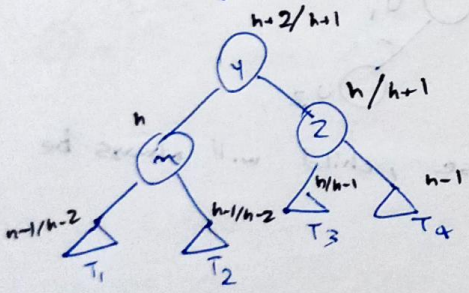
→ y : child of z having larger height
(may or may not be in the path)
(child of y can't have same height as z is imbalanced)
does not exist in path of deleted node

→ x : child of y having larger height
(both child of y can have same height)

→ Cases

I)

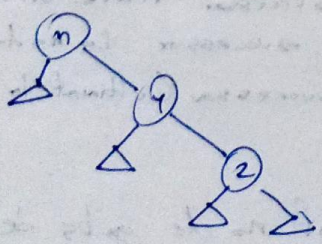




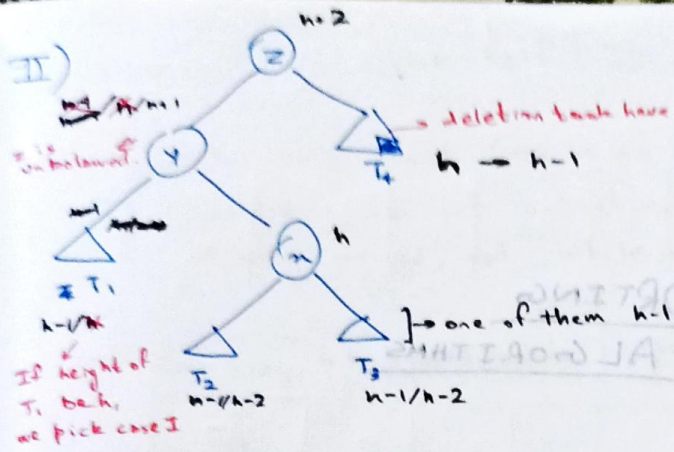
→ if height of y become h+2, it's all cool

→ but if it's h+1, we do check for imbalance again for ancestor of y and do the same.

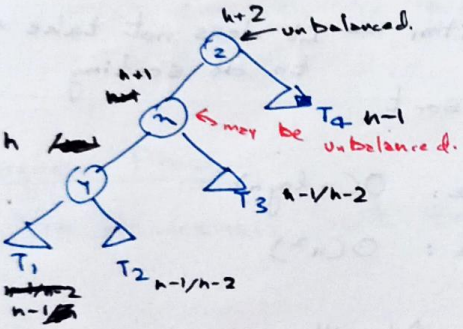
→ symmetric case



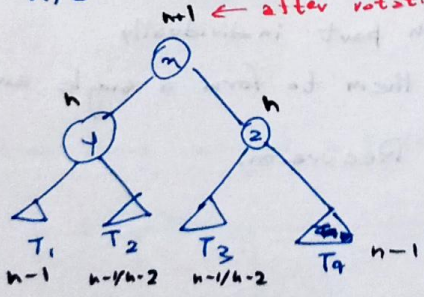
II)



a) Rotation x, y



b) Rotation x, z



→ after these rotation, continue going up checking as height decreases.