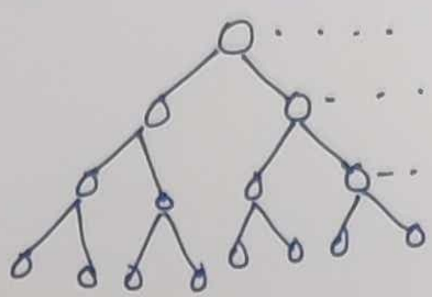
Complete Binary tree.



- level i has 2^i nodes
- In a tree of height h.
  - # leaves $= 2^h$
  - # of internal nodes $=$ # leaves -1
    $= (1+2+2^2+\cdots+2^{h-1} = 2^h-1)$
  - Total # of nodes $= 2^{h+1}-1 = n$

$\Rightarrow$ In a tree of nodes 'n'; # leaves

$$\Rightarrow 2^h \cdot 2 - 1 = n$$
$$\Rightarrow 2^h = (n+1)/2$$
$$\text{Height} = \log_2(\#\text{ leaves})$$

Binary tree :-

Binary tree of height "h"

$\rightarrow$ at most 2^i nodes per level

$\rightarrow$ at most $2^{h+1}-1$ nodes.

$$\Rightarrow \quad n \leq 2^{h+1}-1$$
$$h \geq \log_2 (n+1)/2$$

- Searching -

Search { Linear, Binary }

Binary Search :-

Low = 1, mid = $\lfloor \frac{low + high}{2} \rfloor$
high = n

k = key(mid) : found k
k > key(mid) : recursively search from mid+1 to high

Binary Search (A, k, low, high)

```
if low > high
    return Null
else
    mid = ⌊(low + mid)/2⌋
    if k = A(mid)
        return 'mid'
    else if k > A[mid]
        Binary Search (A, k, mid+1, high)
    else
        Binary Search (A, k, low, mid-1)
```

$$T(n) = \begin{cases} c & , \text{if } n < 2 \\ T(n/2) + c \end{cases}$$

$$\boxed{T(n) = O(\log n)}$$

1

Why we prefer Binary Search Over Linear Search ??

Ex:-

Binary Search Trees (BSTs)

BST is built is such a fashion, that

- x is node in a BST
- y is in left subtree of x
- z — right — x

$x.key \geq y.key$

$z.key \geq x.key$

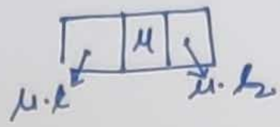
Search (x, u) : looking for value x, with parent node index u.

```
if u = Nil or x = u.key ] Base
    return u

else if x < u.key
    return Search(x, u.l)
else
    return Search(x, u.r)
```

$\underline{\underline{O(h)}}$

Tree Traversal

Inorder: LPR
Preorder: PLR
Postorder: LRP



Inorder(u):

if u ≠ Nil
- inorder (u.l)
- Print (u.key)
- inorder (u.r)

: initially u is root.

O(n)

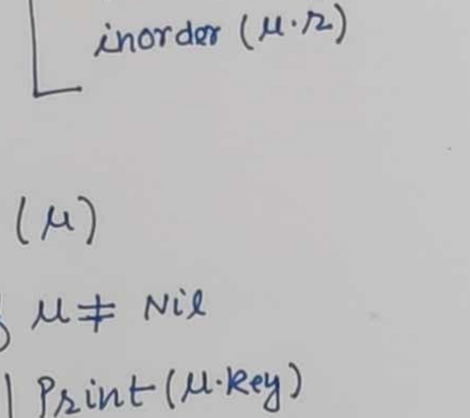
Preorder (u)

if u ≠ Nil
- Print (u.key)
- Preorder (u.l)
- Preorder (u.r)

Postorder

if u ≠ Nil
- Postorder (u.l)
- Postorder (u.r)
- Print (u)

Ex:-



Inorder: gives sorted

: 10, 20, 25. 26, 27, 28, 30. 40

: 30, 20, 10, 25, 22, 27, 26. 28, 40

: 10, 22, 26, 28, 27, 25, 20, 40, 30

Finding Minimum

Min(u)

Recursive

if $u.l = Nil$
  return u
else
  return Min($u.l$)

find

Iterative Min(u)

if $u = Nil$
  return "Empty tree"
while ($u.l \neq Nil$)
  $u \leftarrow u.l$
return u

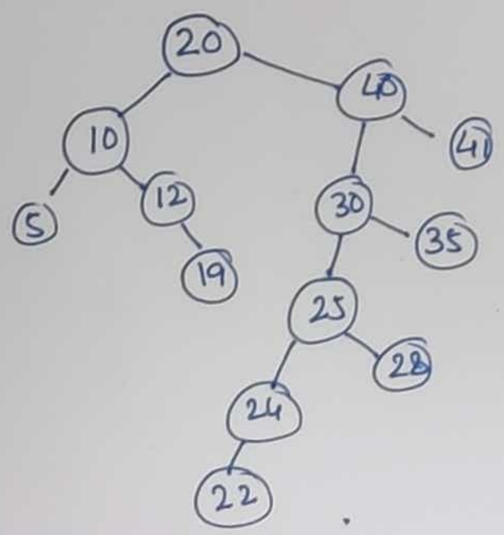
$O(h)$

Max(u)

if $u.r = Nil$
  return u
else
  return Max($u.r$)

Successor: successor of 'u' is a next element which will occur after 'u' if we sort the numbers.

Succ(19) = 20
succ(20) = 22



Successor (u)

if $u.r \neq$ Nil
    return min($u.r$) } case-1

$O(h)$

Case-2
$v \leftarrow u.p$
While $v \neq$ Nil & $u = v.r$
    $u \leftarrow v$
    $v \leftarrow v.p$
return v

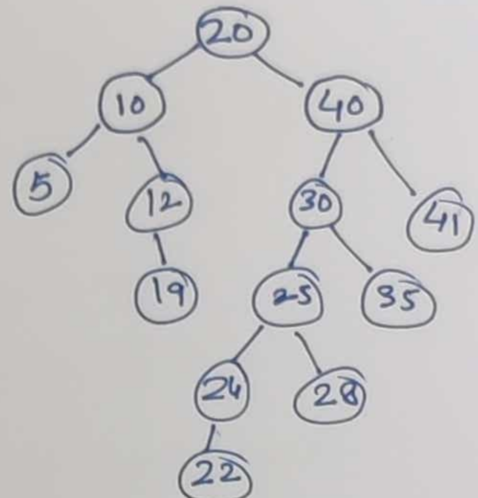
Case-1 :- right subtree of 'u' is non-empty, then successor of 'u" is the left most element of right subtree.

Case 2 :- When ~~white~~ right subtree of u is empty, then we need to find out the ancestor of u whose left child was u.

Predecessor

Defn:- Predecessor(u) is the biggest key ~~in left subtree of (u) which is~~, say $Pr(u) = v$. means.
v.key the biggest key $< u$.key.

Ex:-



Ex:. Pred (20) = 19
Pred (12) = 10
Pred (22) = 20

Case-1:- $u.l$ is non-empty

Pred (u) = Maximum ($u.l$)

Case-2: $u.l$ is empty

* go up the tree until current node is right child, Pred (u) is the parent of current node.
* if we can't go further u is the smallest element.

Predecessor (u)

```
if u.l ≠ Nil
    return maximum (u)
else
    v ← u.p
    While v ≠ Nil & u = v.l      : go up
        u ← v
        v ← v.p
    return v
```

O(h)

Insertion of a node:-

Recursive Procedure:

Insert (u, v)

```
if u = Nil                  } Base case
  | u.Key ← v.Key
else
  | if v.Key < u.Key                         O(h)
  |   | return Insert(u.l, v)
  |   else
  |   | return insert(u.r, v)
```

* It is like searching

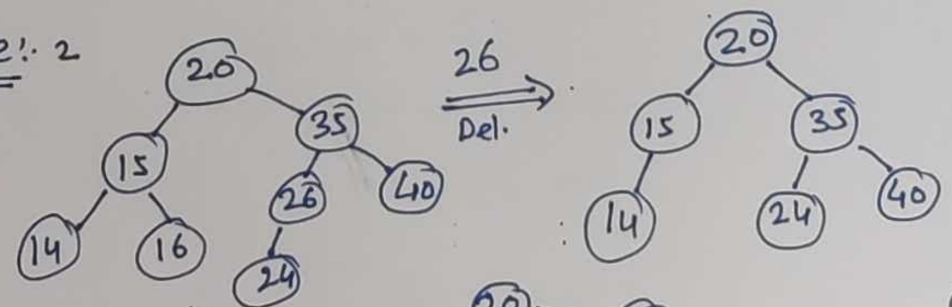
Deletion of node

Case-1. Node to be deleted has no child.
Delete node and replace it by nil.

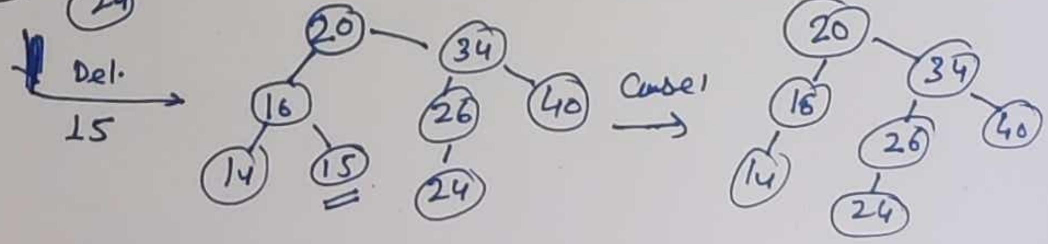
Case: 2. node has only one child, find immediate ancestor (Predecessor) and attach child with ancestor.

Case: 3. Two chil.
find out its successor, which has no left child and exchange it.

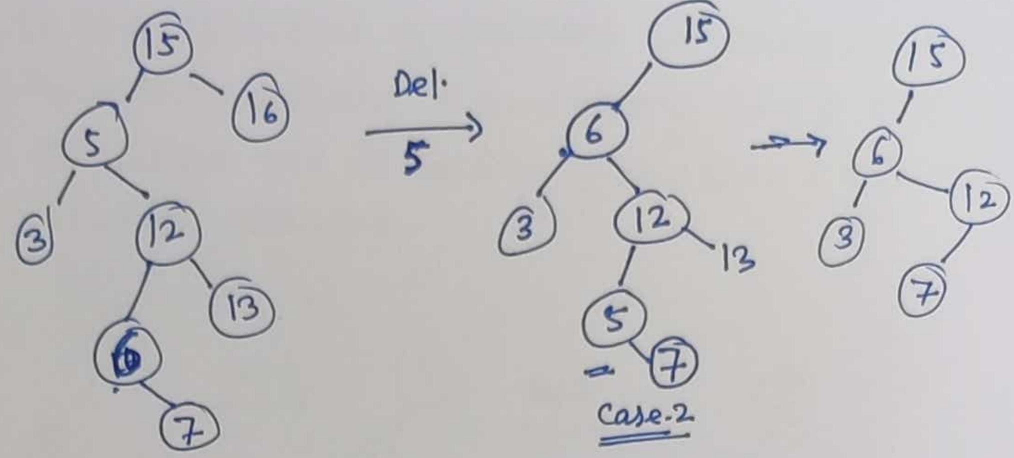
Case: 2



Case: 3



Ex:-



Delete (u)

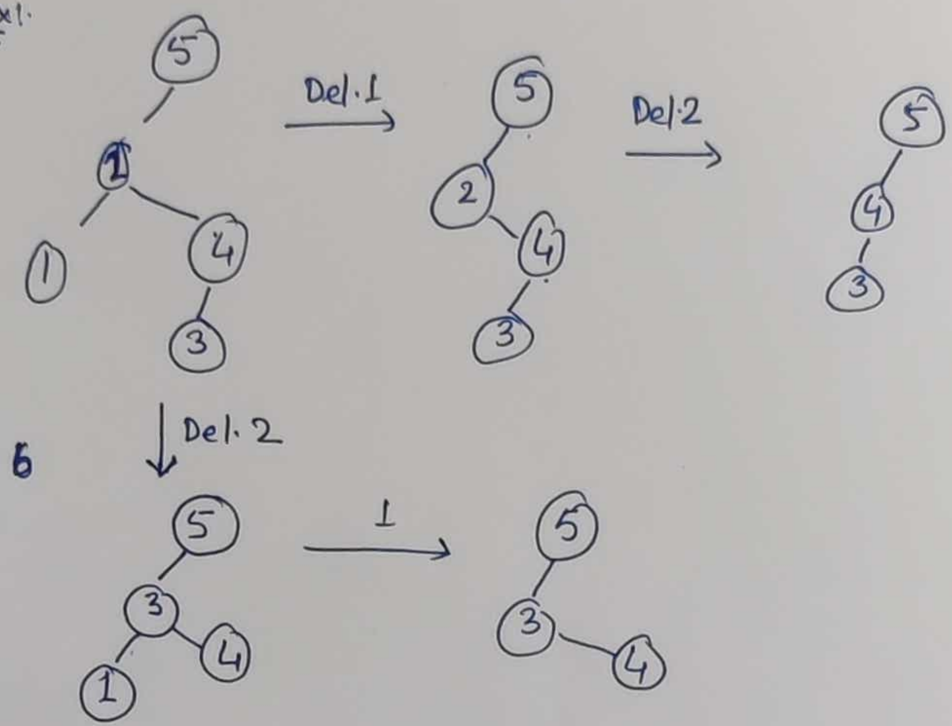
```
1 { if u.l = nil or u.r = nil
  |     v ← u
  |  else
  |     v ← successor(u)

2 { if v.l ≠ Nil
  |     τ ← v.l
  |  else
  |     τ ← v.r

3 { if τ ≠ Nil
  |     τ.p ← v.p

4 { if v.p = Nil
  |     T.root ← τ
  |  else if v ← (v.p).l
  |     |  (v.p).l ← τ
  |     else
  |        (v.p).r ← τ
```

$O(h)$

```
if v ≠ u
   u.key ← v.key
return v
```

<u>Quest</u>:- Is the operation of deletion is "commutative" such that deleting 'x' and then y from a BST leaves the same tree as deleting y and then x? Give counter example.

<u>Ex1.</u>



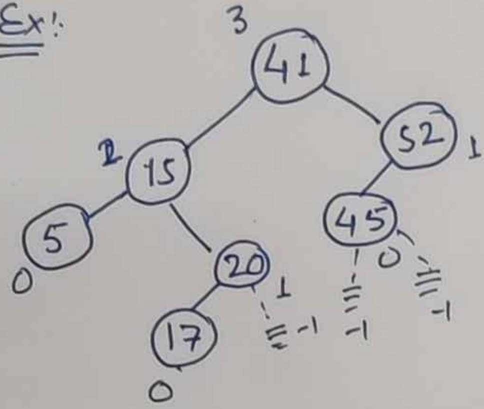
AVL Tree : Self-balancing (dynamically balanced) BST.

- Named after: Adelson-Velsky-Landis (1962)
  (Soviet Mathematician)

Height of a node :- Length of longest path from that node down to a leaf.

= max (height of left child, right child) + 1

Ex:



CONCEPT OF AVL TREE :-

- for each node (μ), the height of left subtree & right subtree of μ is differ by at most 1.

$$\boxed{|h_l - h_r| \leq 1}$$

Height of AVL Tree : worst case:- when right subtree has height one more than left subtree.

Recursive Definition :

n_h : min # nodes of height h.

N_{h-1} : # of nodes in Tree of ht $h-1$

Basecase: $N_1 = 1$

$N_2 = 2$

$N_h = 1 + N_{h-1} + N_{h-2}$, $h \geq 3$

$> 1 + 2N_{h-2}$

$> 2N_{h-2}$

$$N_h > 2N_{h-2} \quad -①$$

N_h at least doubles ~~at~~ each time h increases by 2. means N_h grows exponentially

$$N_h > 2^{\frac{h}{2}-1}$$

$$\log_2 N_h > \frac{h}{2} - 1$$

$$h < 2 + 2\log N_h$$

$$\boxed{h = O(\log n)} \qquad \boxed{N_h = n}$$

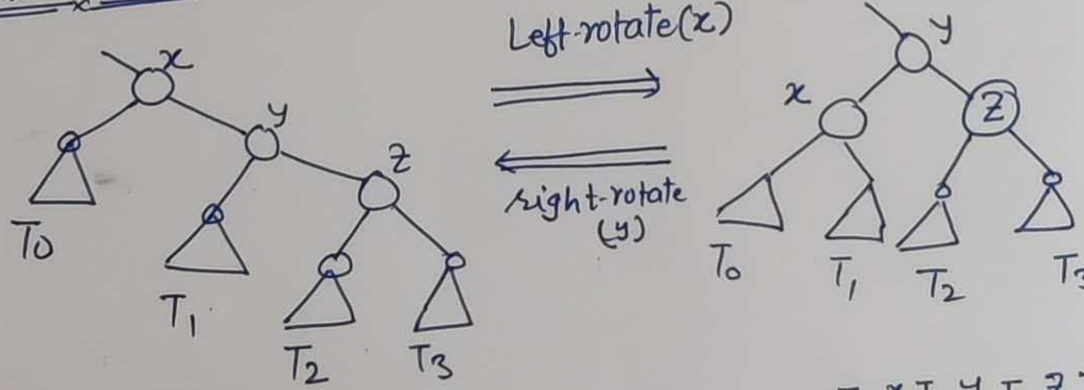
Illustration

$N_3 = 2N_1 = 2 \times 2^{3/2+1}$
$N_4 = 2N_2 = 2 \times 2^{4/2-2}$
$N_5 = 2N_3 = 2 \times 2^{\lfloor 5/2 \rfloor - 2}$
$N_6 = 2 \times N_4 = 2 \times 2^{6/2-1}$
$= 8$

Insertion :

- Insert like BST
- Maintain AVL Properties.

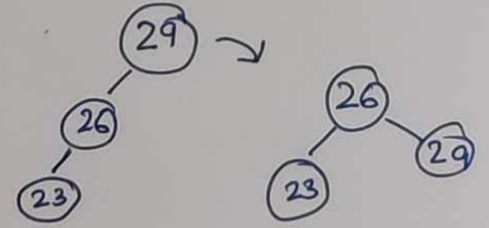
Rotation :-



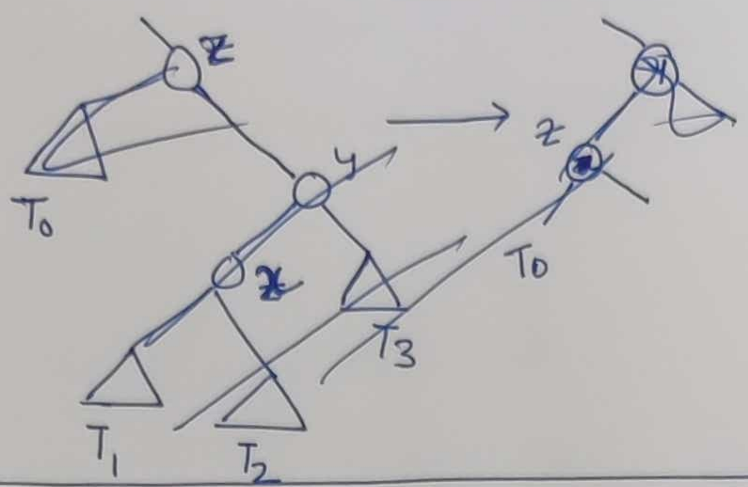
Inorder Traversal: $T_0\, x\, T_1\, y\, T_2\, z\, T_3$ $\qquad$ $T_0\, x\, T_1\, y\, T_2\, z\, T_3$

BST Prop. Maintained

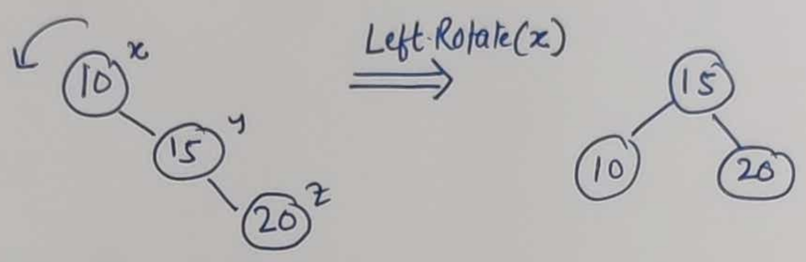
Ex:



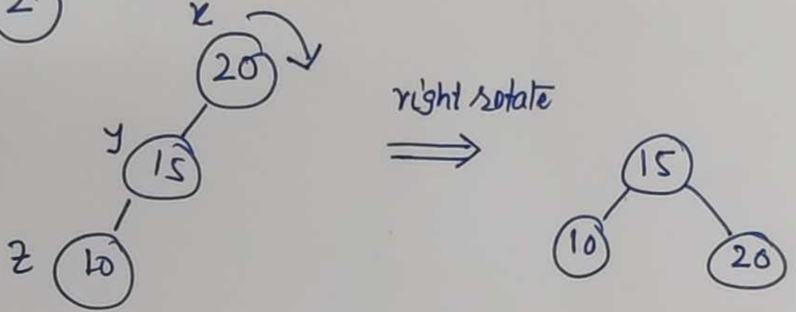
Double Rotate



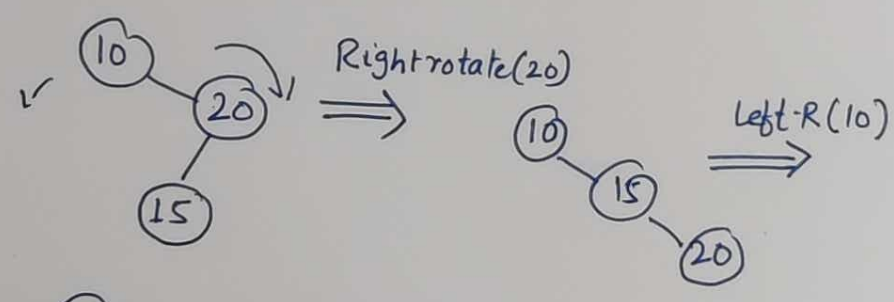
Cases: 1



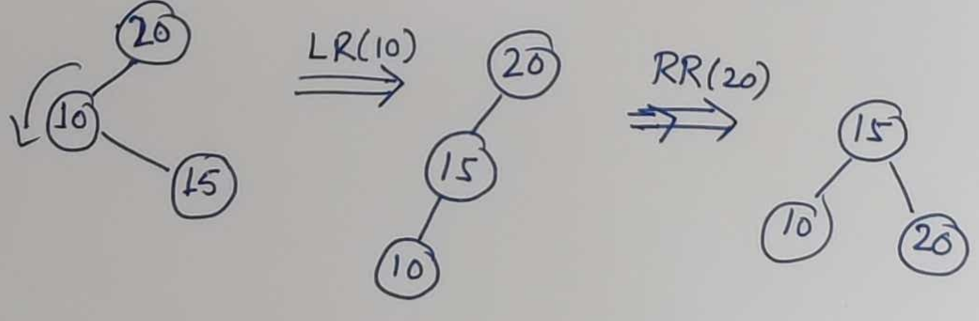
2.



3



4



AVL TREE Example

Insert 'n' items – $\Theta(n \log n)$

Inorder traversal – $\Theta(n)$.

Ex:- 4, 10, 2, 1, 0, -3