

# DATA STRUCTURES AND ALGORITHMS

## TASK 7.1P

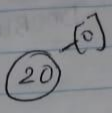
### AVL TREES

AVL TREES

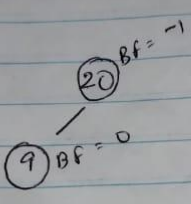
VALUES TO INSERT

20, 9, 3, 7, 5, 8, 25, 30, 15, 6, 17

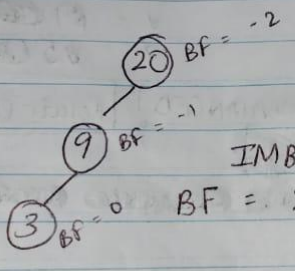
① Inserting 20:



② Inserting 9:

  
V = 20, Z = 9

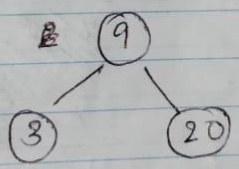
③ Inserting 3:

  
V = 20, Z = 3

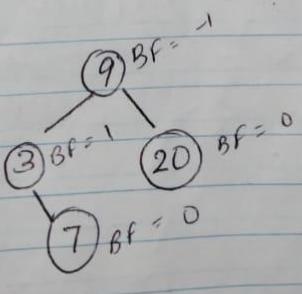
left heavy } IMBALANCED! Rotate Right

BF = 2

So,

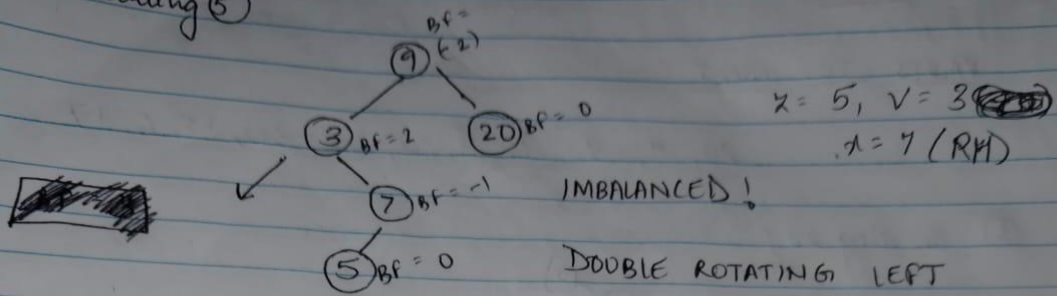


④ Inserting 7:

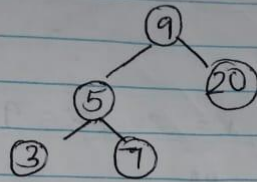
  
Z = 7, V = 9  
X = 3

left heavy

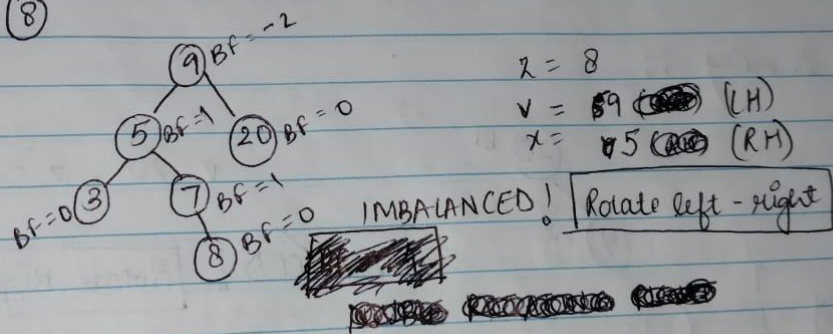
⑤ Inserting 5



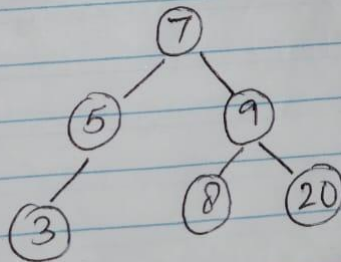
So,



⑥ Inserting 8

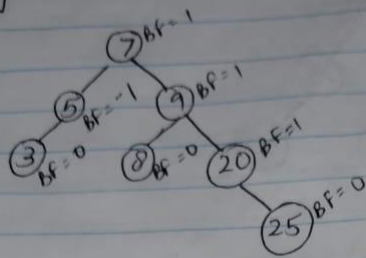


So,



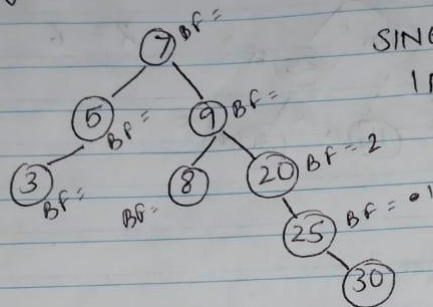
~~x = 55~~

⑦ Inserting 25:



$x = 25$   
 $y = 9$  (RH)  
 $z = 20$  (RH)

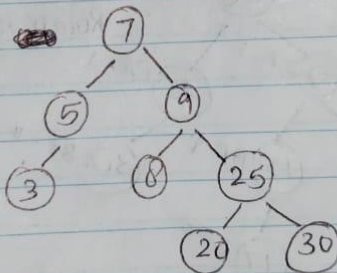
⑧ Inserting 30:



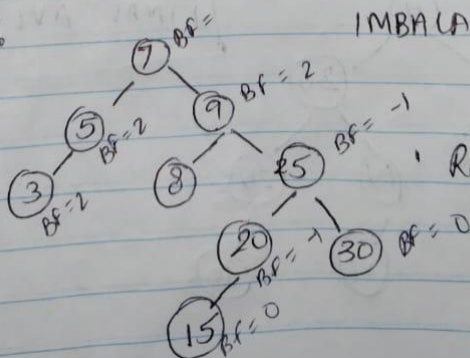
SINGLE ROTATING LEFT  
 IMBALANCED!

$x = 30$   
 $y = 20$  (RH)  
 $z = 25$  (RH)

30,



⑨ Inserting 15:

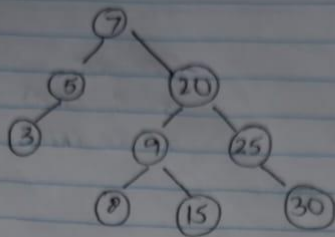


IMBALANCED!

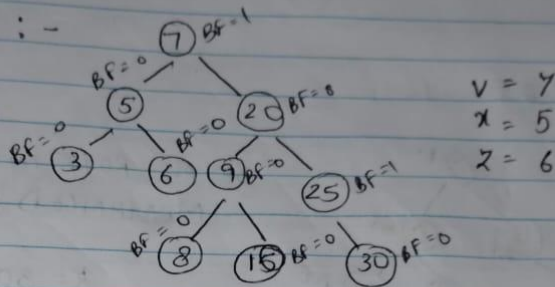
Rotate right-left

$y = 9$  (RH)  
 $x = 25$  (LH)  
 $z = 15$

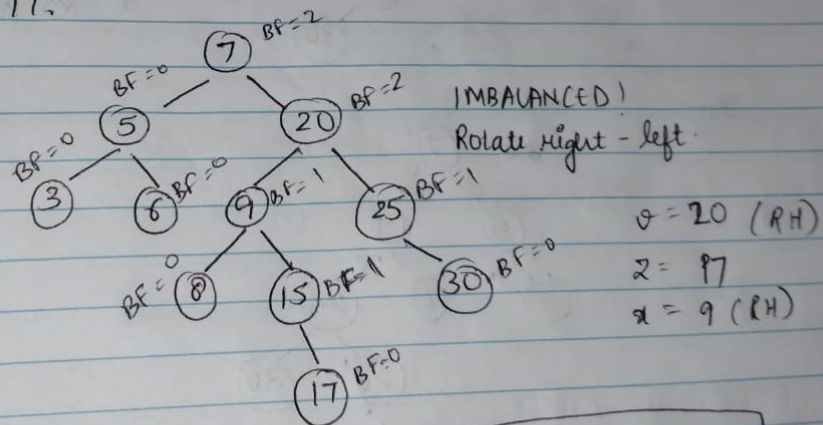
So,



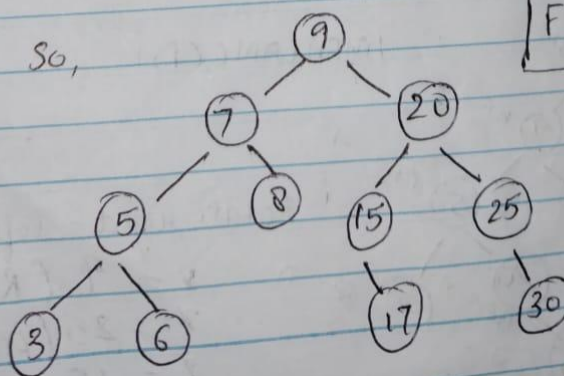
(10) Inserting 6 :-



(11) Inserting 17:



So,

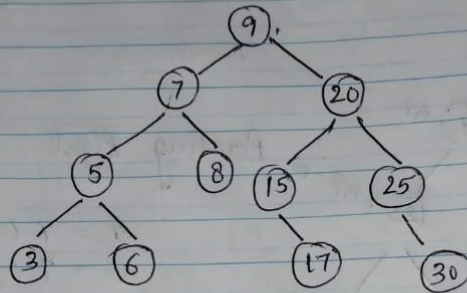


FINAL AVL TREE

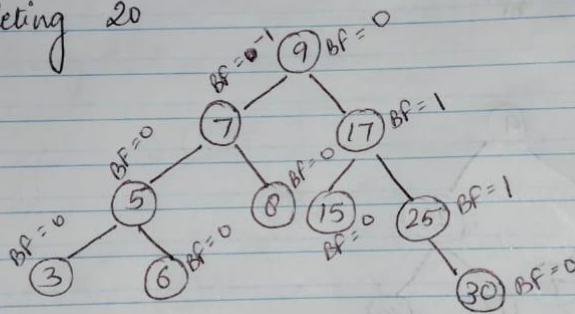
## DELETION

20, 15, 8, 25, 30, 9, 17, 5, 6, 3, 7

The final built AVL TREE :



① Deleting 20

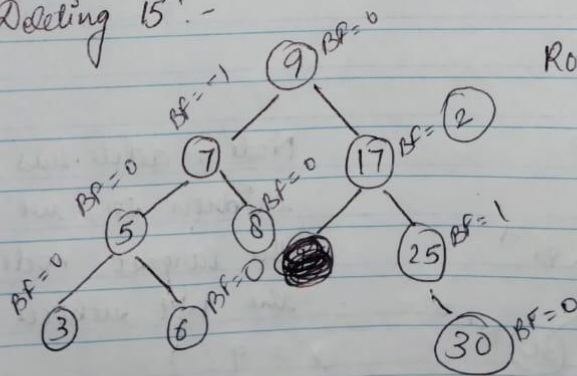


~~X = 20~~ X = 9

~~V = 20~~ V = 9

Z = 20

② Deleting 15 :-

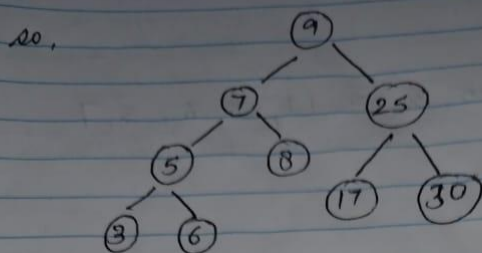


Rotating left

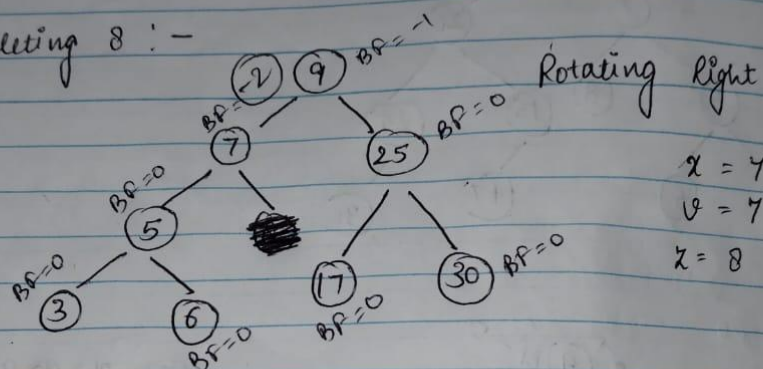
X = 15

V = 17 (RH)

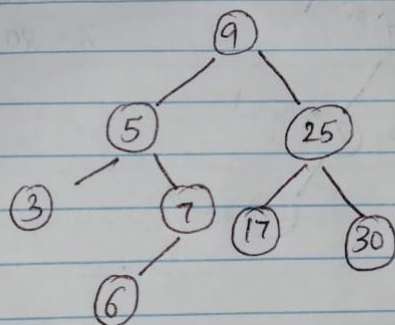
X = 25 (RH)



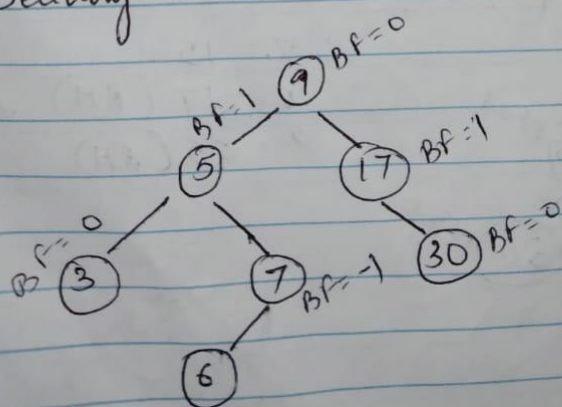
③ Deleting 8 :-



20,



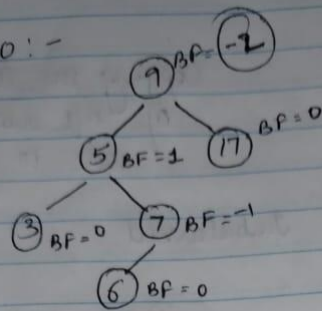
④ Deleting 25 :-



Node to delete has two children so, we find the largest node in the left subtree.

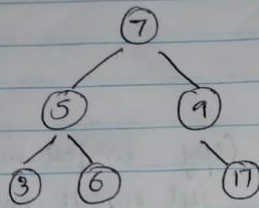
$x = 9 (-)$   
 $y = 9 (-)$   
 $z = 25$

⑤ Deleting 30 :-

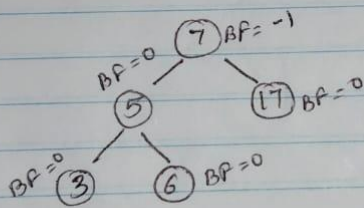


Rotating Right  
 $x = 17$  (-)  
 $y = 9$  (LH)  
 $z = 30$

so,



⑥ Deleting 9 :-

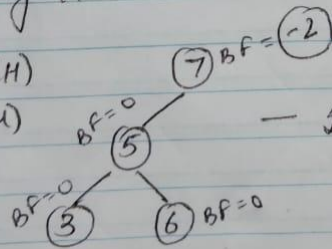


Node to delete has no child,  
 Set parent of deleted node to  
 right child of deleted node.

$x = 7$  (LH)  
 $y = 7$  (LH)  
 $z = 9$

⑦ Deleting 17 :-

$x = 7$  (LH)  
 $y = 7$  (LH)  
 $z = 17$

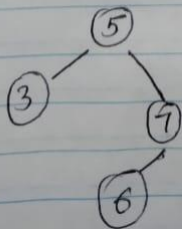


— Imbalanced

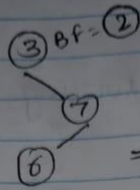
The node has no children

Single rotate right.

So,



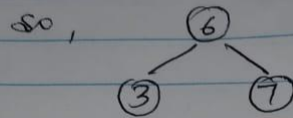
⑧ Deleting 5 :-



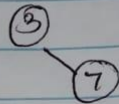
Copying the largest value of left subtree into node to delete

= Imbalanced

Rotate left  $\times = 5$



⑨ Deleting 6 :-



Copy largest value of left subtree into node to delete.

⑩ Deleting 3 :-



⑪ Deleting 7 :-

#### PSEUDOCODE:

function rebalanceBST(root):

values = [] // Step 1: Create empty list

collectInOrder(root, values) // Step 2: Fill list with sorted values

```
return buildTree(values, 0, length(values) - 1) // Step 3: Build balanced BST
```

```
function collectInOrder(node, list):
```

```
    if node is null:
```

```
        return
```

```
    collectInOrder(node.left, list)
```

```
    list.add(node.value)
```

```
    collectInOrder(node.right, list)
```

```
function buildTree(arr, left, right):
```

```
    if left > right:
```

```
        return null
```

```
    mid = (left + right) // 2
```

```
    root = new Node(arr[mid])
```

```
    root.left = buildTree(arr, left, mid - 1)
```

```
    root.right = buildTree(arr, mid + 1, right)
```

```
    return root
```

## Description

This algorithm takes any binary search tree (BST), even if it is highly unbalanced, and builds a new BST that is balanced and has height around  $O(\log n)$ . The process works in two main phases.

First, the algorithm goes through the tree using in-order traversal and records each node's value into a list. This ensures the values are stored in sorted order because in-order traversal of a BST always visits nodes from smallest to largest.

Next, the algorithm builds a new tree from that sorted list by repeatedly picking the middle value to be the root. The left half of the list becomes the left subtree, and the right half becomes the right subtree. This recursive splitting keeps the tree as balanced as possible at every level.

Since each node is visited only once during traversal and once during reconstruction, the total time taken is linear in the number of nodes. The resulting tree keeps the binary search property and is balanced in structure, making operations like search, insert, and delete efficient.

## Why the Algorithm Takes Linear Time ???

The total time complexity is  $O(n)$  because:

### In-order Traversal

- Each node in the tree is visited **once** during the in-order traversal.

- For every node, we perform a **constant amount of work** (appending to a list).
- So, for  $n$  nodes, the traversal takes:

$O(n)$

### Building the Balanced BST

- The recursive function `buildBalancedBST` also creates **one node per list element**.
- At each step:
  - We select the middle index ( $O(1)$ ),
  - We create a node ( $O(1)$ ),
  - Then we recurse on left and right halves.
- Since no element is visited more than once, and each node is constructed with **constant work**, this step also takes:

$O(n)$

### Total Time

Adding the two steps:

- In-order traversal:  $O(n)$
- Tree construction:  $O(n)$

$\Rightarrow \text{Total Time} = O(n) + O(n) = O(n)$

No sorting is needed

No rotations or AVL balancing logic

Each element is processed a fixed number of times