Lecture 30-31

- Balanced Binary Tree (AVL Tree)

IT205: Data Structures (AY 2023/24 Sem II Sec B) — Dr. Arpit Rana

Issues with Binary Search Tree

Average search time in a binary search tree

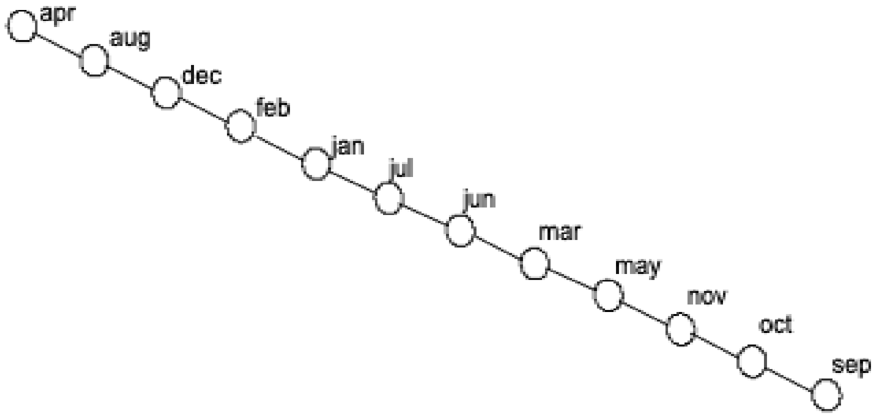
$$\Gamma = \frac{\sum_{i=1}^{n} \tau_i}{n}$$

- τ_i = Number of comparisons for the i^{th} element
- n = Total number of elements in the binary search tree

A binary search tree should be with a minimum value of average search time (Γ).

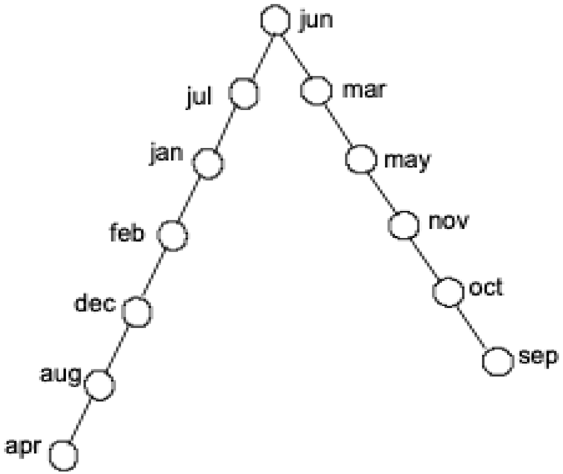
Average Search Time Calculation: Case 1

A skewed binary tree obtained from lexicographical order of data

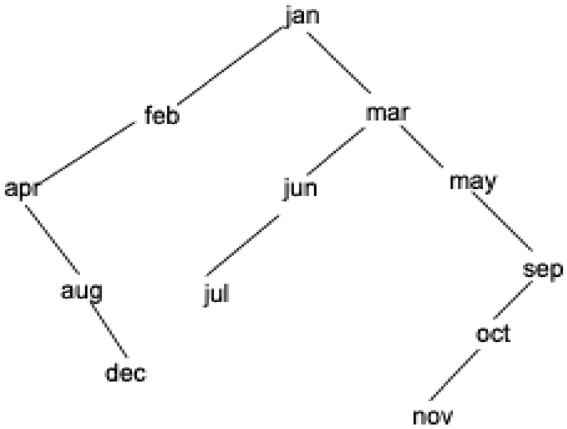


$$\Gamma = (1 + 2 + \ldots + 12) / 12 = 6.5$$

Average Search Time Calculation: Case 2



A binary search tree (half skewed version)

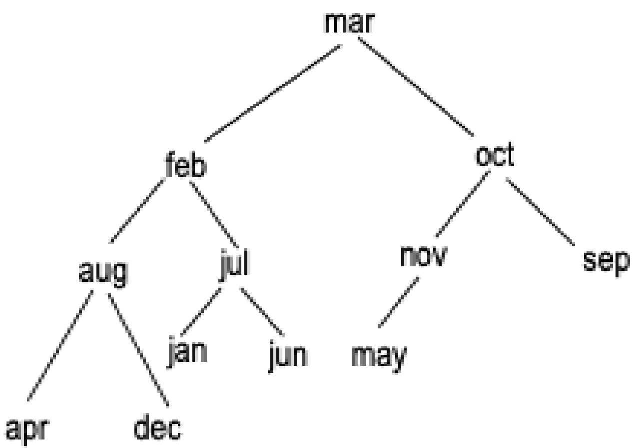


A binary search tree (obtained by inserting the data into the order of months).

Γ = [(1 + 2 + . . . + 7) + (2 + 3 + . . . + 6)] / 12 = 4

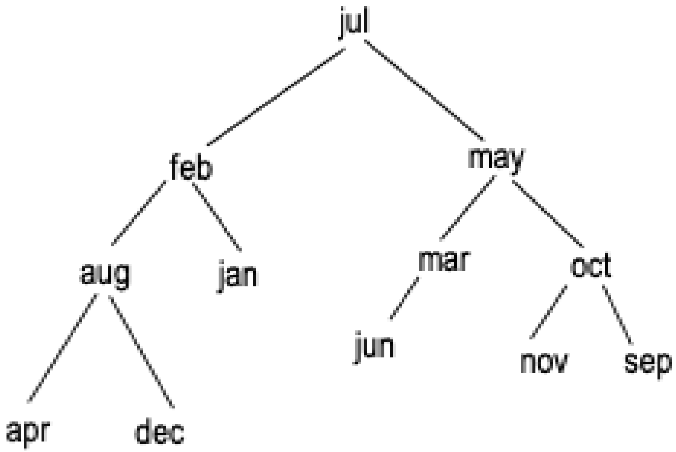
Γ = [(2 + . . . + 5) + (3 + 4) + (1 + . . . + 6)] / 12 = 3.5

Average Search Time Calculation: Case 3



$\Gamma = 3.08$

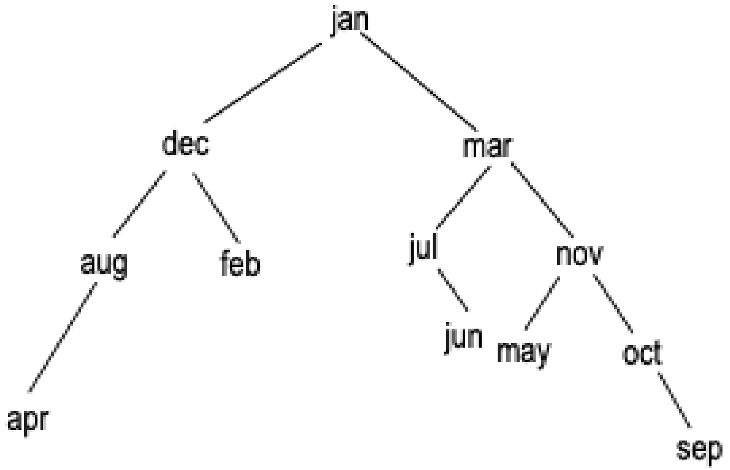
(1+2+3+4)+4+(3+4)+4+(2+3)+(3+4)/12



$\Gamma = 3.08$

(1+2+3+4)+4+(3)+(2+3+4)+(3+4)+(4)/12

Average Search Time Calculation: Case 4



A binary search tree obtained by a special technique

it means that when we chose balanced binary tree it reduces avg search time

Γ = 3.16

(1+2+3+4)+(3)+(2+3+4+5)+(3+4)+4/12

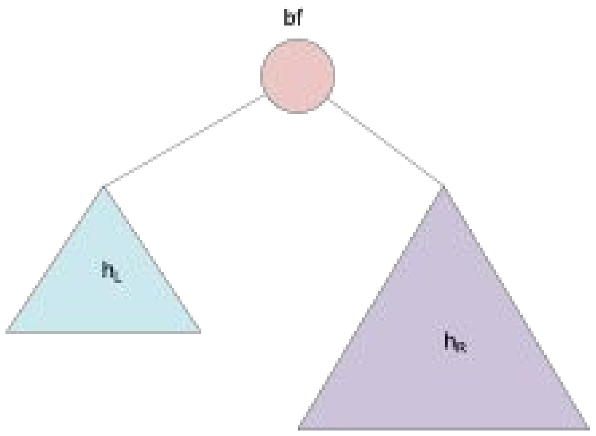
Source: https://www.prepbytes.com/blog/tree/expression-tree-in-data-structure/

Finding the Best BST

How to find a binary search tree with a minimum value of average search time?

- Height balanced binary search tree a.k.a AVL tree (Adelson–Velsky and Landis)
- Concept of balance factor of a node

$$bf = \text{Height of the left subtree } (h_L) - \text{Height of the right subtree } (h_R)$$

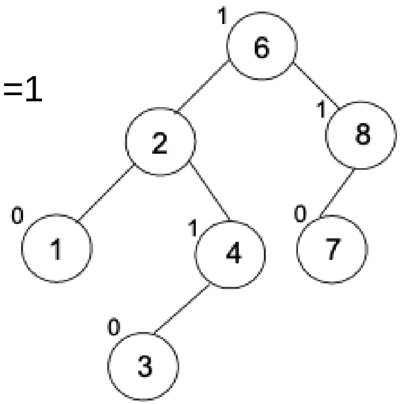


Height Balanced Binary Tree: Definition

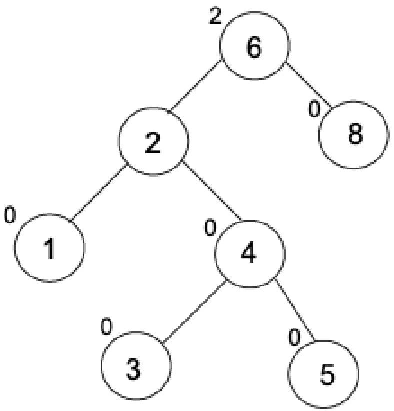
A balanced binary tree (a.k.a. AVL tree) is a binary tree in which the height of the two subtrees of every node never differ by more than 1.

$$bf = | h_L - h_R | \leq 1$$

consider root node height =1



(a) Height balanced



(b) Height unbalanced

Making a Binary Tree Height Balanced

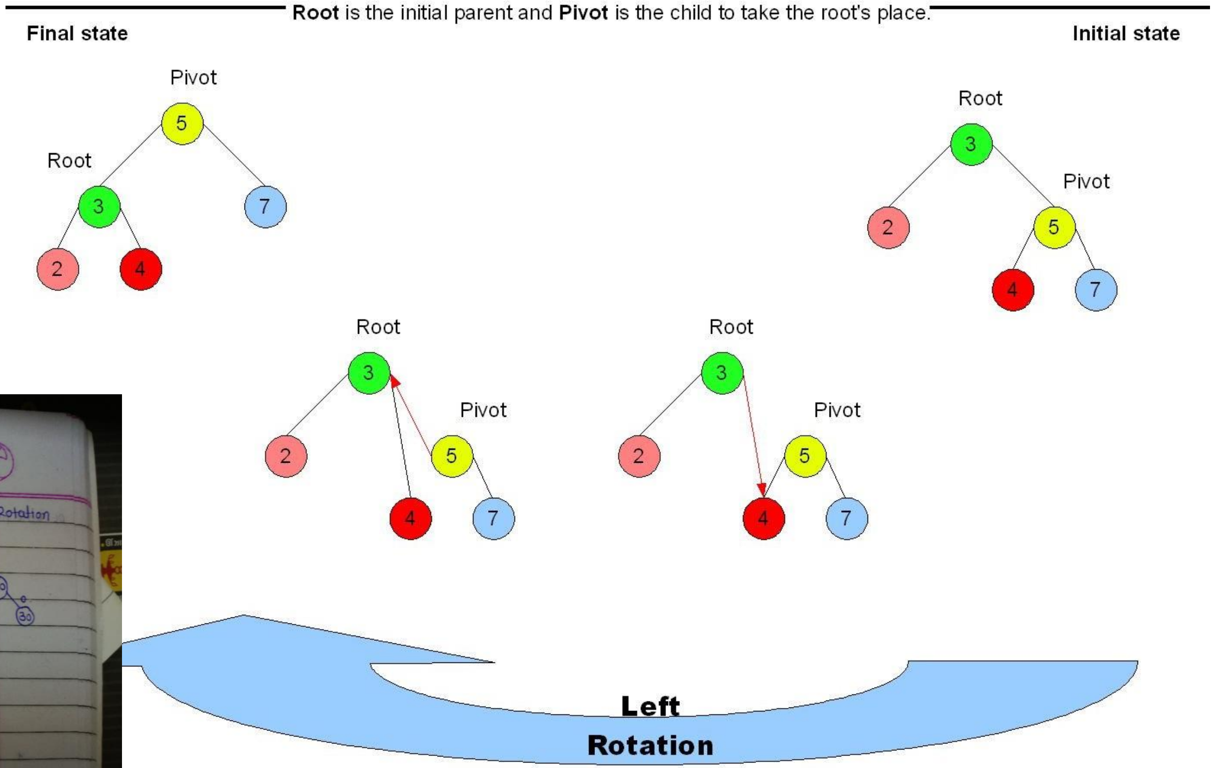
Steps
:

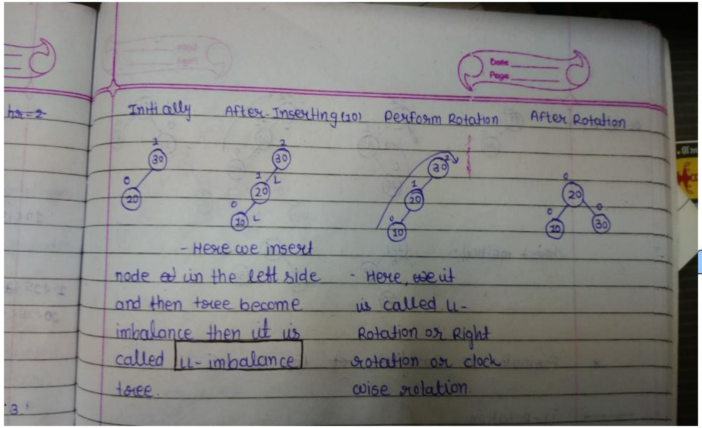
- Insert node into a binary search tree
- Compute the balance factors
- Decide the pivot node
- Balance the unbalance tree through rotations

AVL Rotations: Left Rotation

Left (anti-clockwise) Rotation

```
q = right(p)
hold = left(q)
left(q) = p
right(p) = hold
```

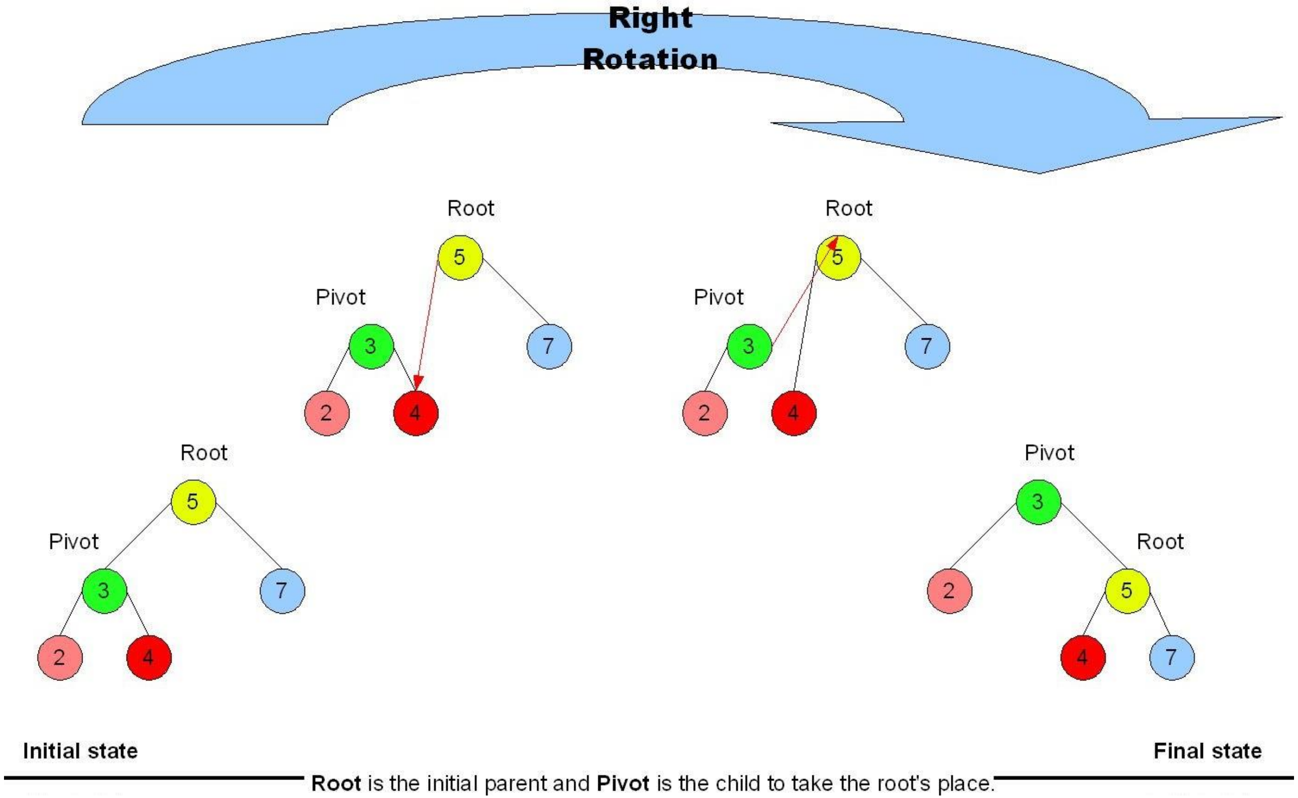




AVL Rotations: Right Rotation

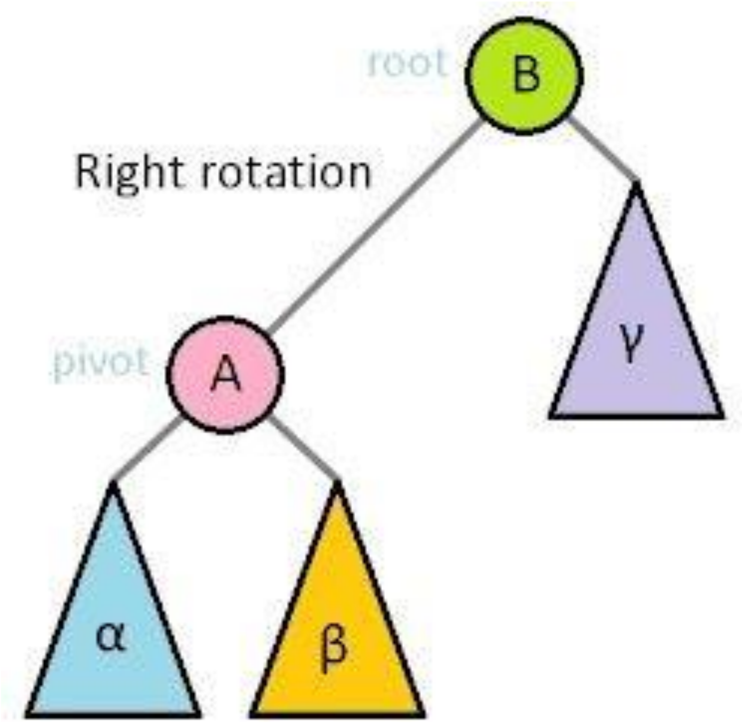
Right (clockwise) Rotation

```
q = left(p)
hold = right(q)
right(q) = p
left(p) = hold
```

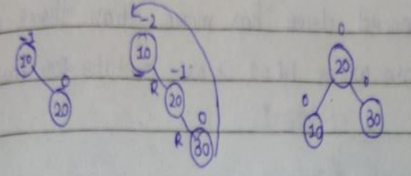


AVL Rotations

- LL (Left) Rotation
- RR (Right) Rotation
- LR (Left+Right)Rotation
- RL (Right+Left) Rotation

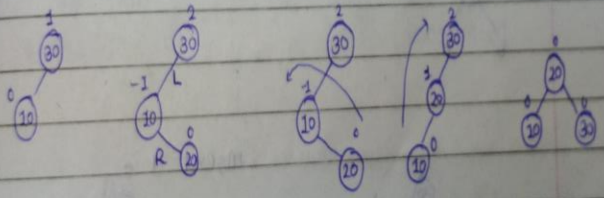


Rotations are done ~~for~~ with only 3 nodes only.



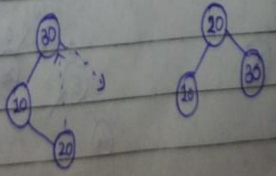
(RR Rotation / left Rotation / ~~anti~~ counter clockwise Rotation)

If only 3 node imbalanced tree and we use any of the rotation it gives same answer.

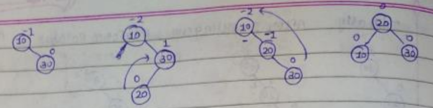


That is also called as double rotation

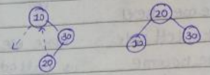
direct method :-



RL-Rotation :-



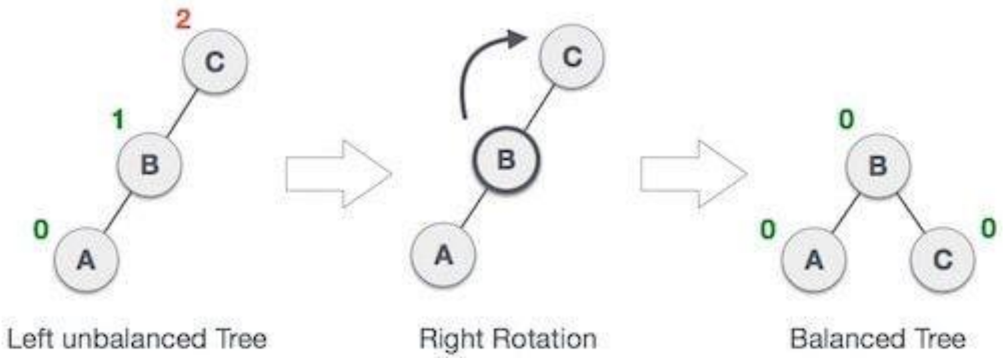
direct method :-



Making a Binary Tree Height Balanced

Right (RR) rotation

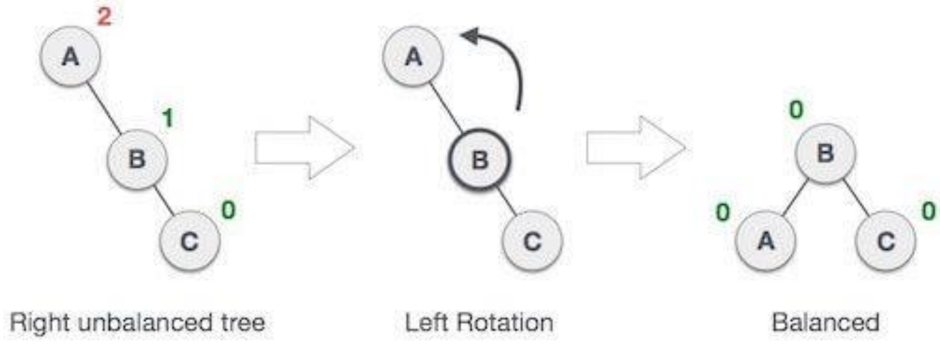
A single rotation applied when a node is inserted in the left subtree of a left subtree. In the given example, node C now has a balance factor of 2 after the insertion of node A. By rotating the tree right (clockwise), node B becomes the root resulting in a balanced tree.



Making a Binary Tree Height Balanced

Left (LL) rotation

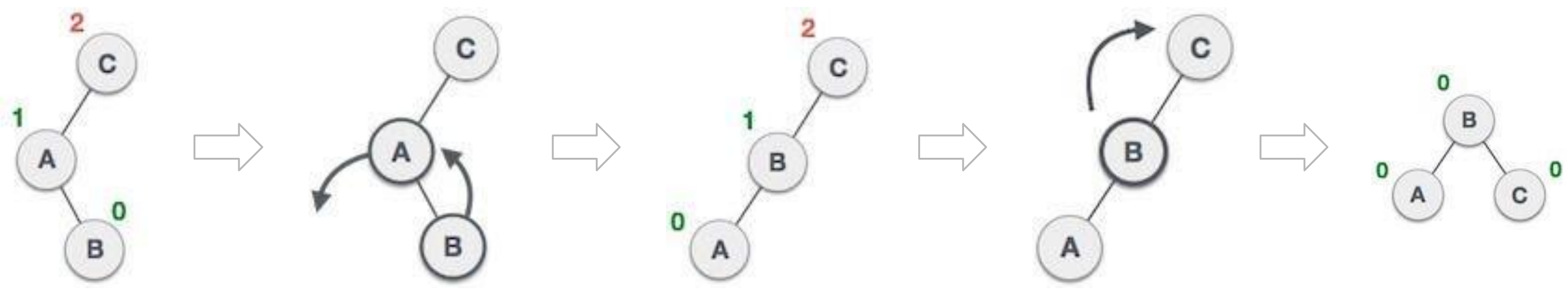
A single rotation applied when a node is inserted in the right subtree of a right subtree. In the given example, node A has a balance factor of 2 after the insertion of node C. By rotating the tree left (anti-clockwise), node B becomes the root resulting in a balanced tree.



Making a Binary Tree Height Balanced

Left-Right (LR) Rotation

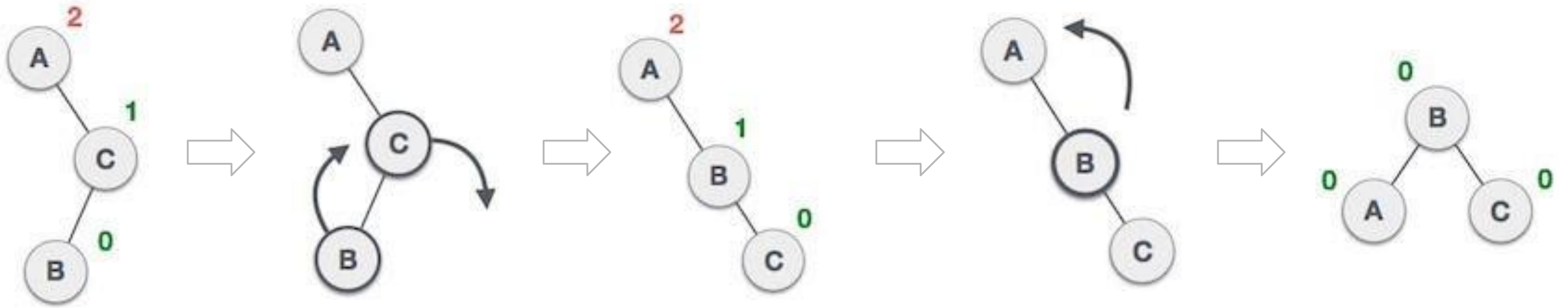
A double rotation in which a left rotation is followed by a right rotation. In the given example, node B is causing an imbalance resulting in node C to have a balance factor of 2. As node B is inserted in the right subtree of node A, a left rotation needs to be applied. However, a single rotation will not give us the required results. Now, all we have to do is apply the right rotation as shown before to achieve a balanced tree.



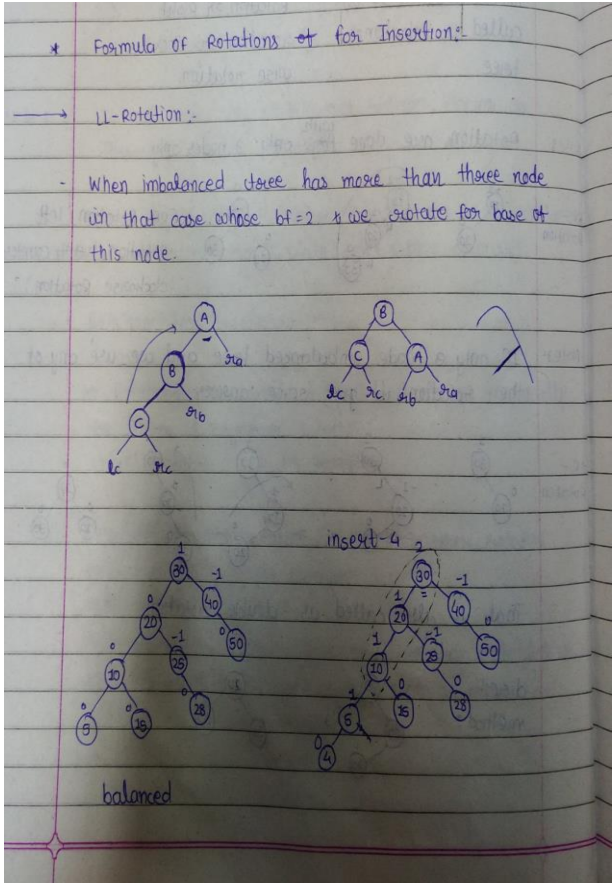
Making a Binary Tree Height Balanced

Right-Left (RL) Rotation

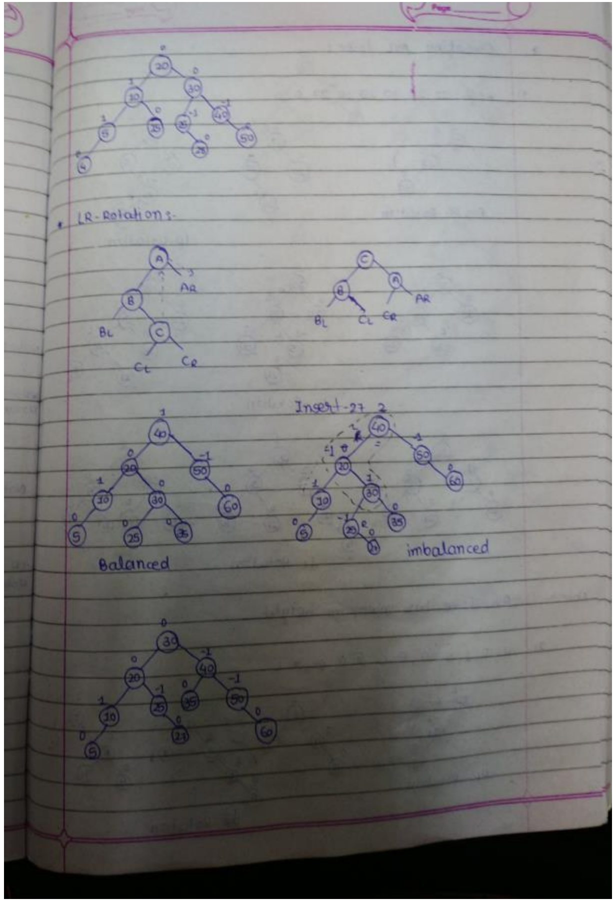
A double rotation in which a right rotation is followed by a left rotation. In the given example, node B is causing an imbalance resulting in node A to have a balance factor of 2. As node B is inserted in the left subtree of node C, a right rotation needs to be applied. However, just as before, a single rotation will not give us the required results. Now, by applying the left rotation as shown before, we can achieve a balanced tree.



LL ROTATION



LR ROTATION

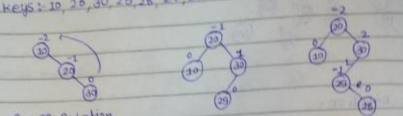


Rotations: When to Use What

```
IF tree is right heavy {
    IF tree's right subtree is left heavy {
        Perform Right Left rotation
    }
    ELSE {
        Perform Single Left rotation
    }
}
ELSE IF tree is left heavy {
    IF tree's left subtree is right heavy {
        Perform Left Right rotation
    }
    ELSE {
        Perform Single Right rotation
    }
}
```

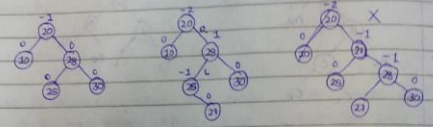
* Creating AVL tree:

1) Keys: 10, 20, 30, 25, 28, 27, 5.

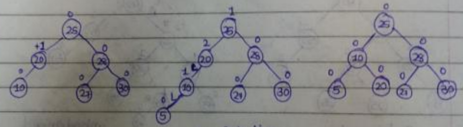


~~R→~~ RR-Rotation

LR-Rotation



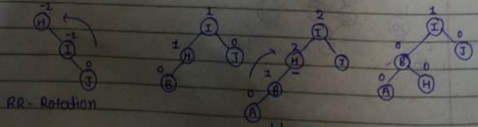
RL-Rotation



LL-Rotation

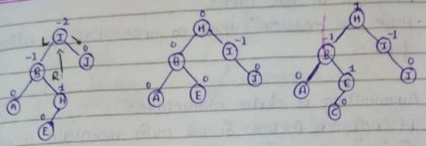
Note:- AVL Tree has minimum height

2) H, I, J, B, A, E, C, F, D, G, K, L

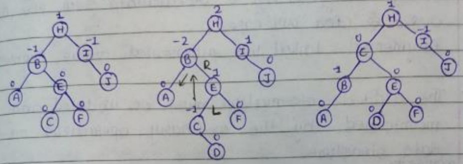


RR-Rotation

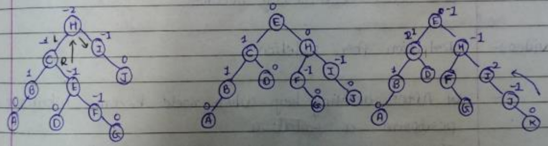
LL ~~RR~~-Rotation



LR Rotation

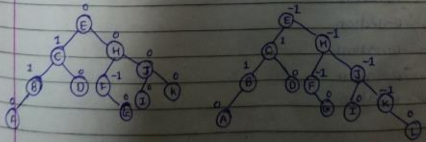


RL-~~LR~~ Rotation



LR-Rotation

RR ~~left~~ Rotation



AVL Tree

Balanced Binary Tree: Property

Maximum height possible in an AVL tree with *n* number of nodes is given by

$$h_{\max} = 1.44 \log_2 n$$

Exercise

Construct an AVL tree with the following elements:

`H, I, J, B, A, E, C, F, D, G, K, L`

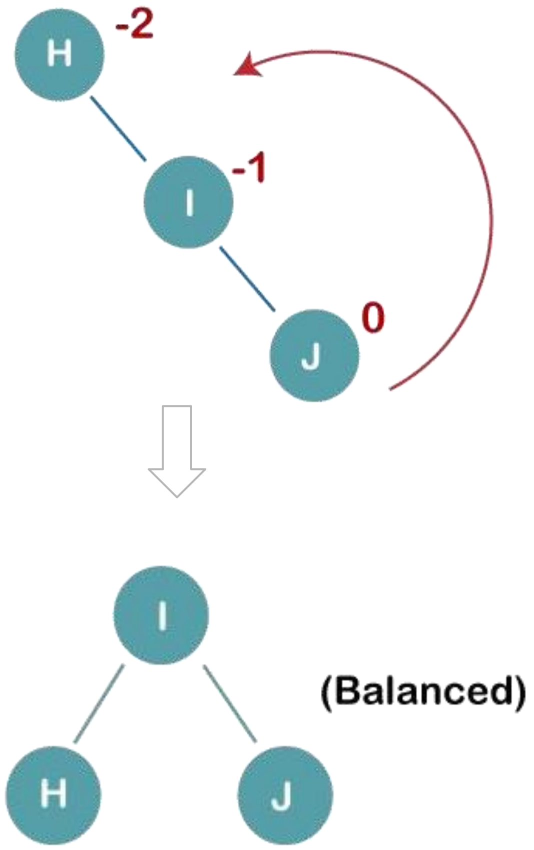
Code - > avl2.cpp

Insertion in an AVL Tree

Construct an AVL tree with the following elements:

<u>H, I, J,</u> B, A, E, C, F, D, G, K, L

- On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2.
- Since the BST is right-skewed, we will perform LL (anti-clockwise) Rotation on node H.

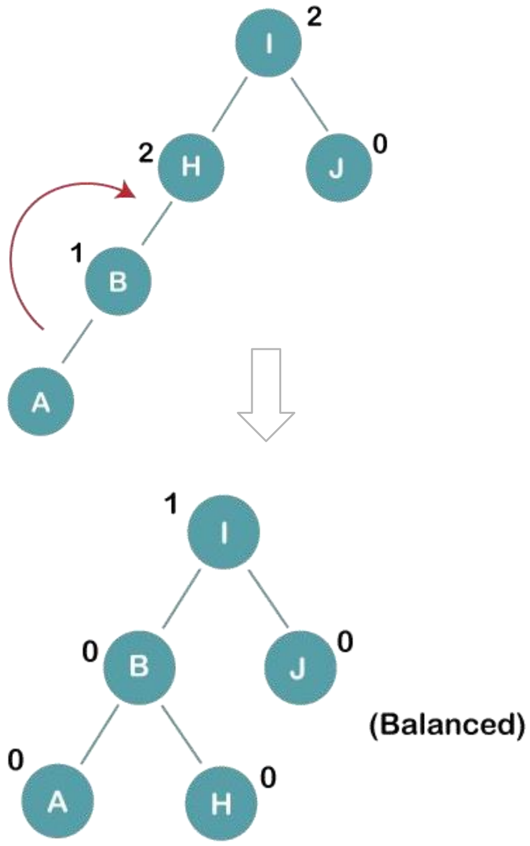


Insertion in an AVL Tree

Construct an AVL tree with the following elements:

`H, I, J,` **`B, A, E, C, F, D, G, K, L`**

- On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H.
- Since the BST from H is left-skewed, we will perform RR (clockwise) Rotation on node H.

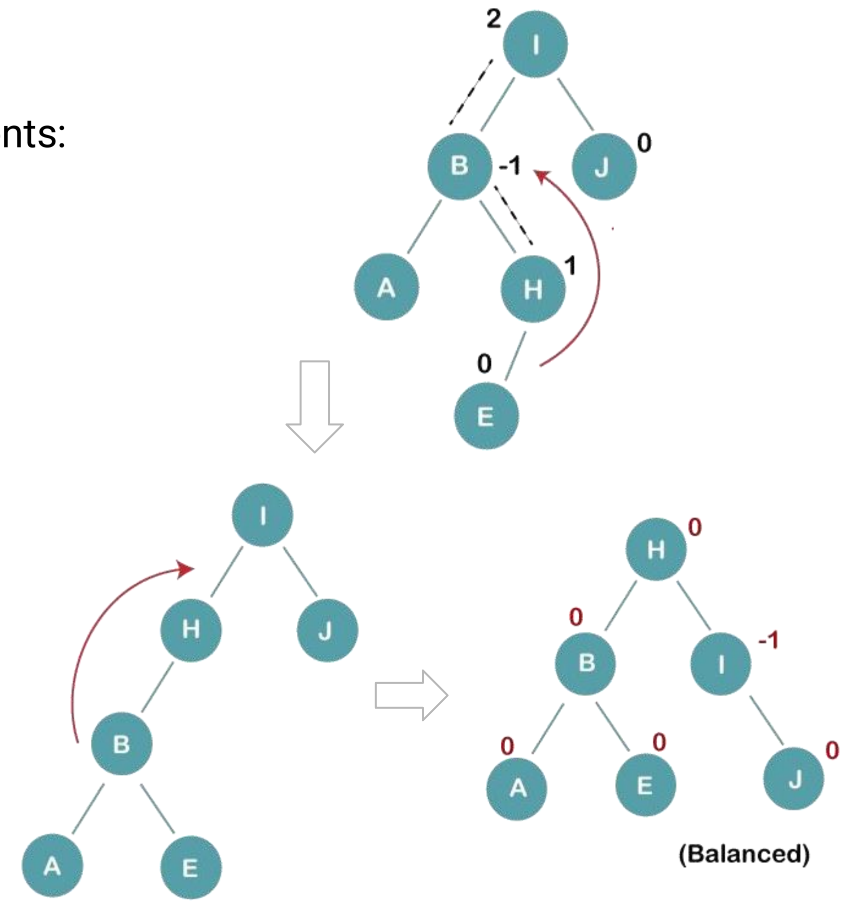


Insertion in an AVL Tree

Construct an AVL tree with the following elements:

H, I, J, B, A, **E**, **C**, **F**, **D**, **G**, **K**, **L**

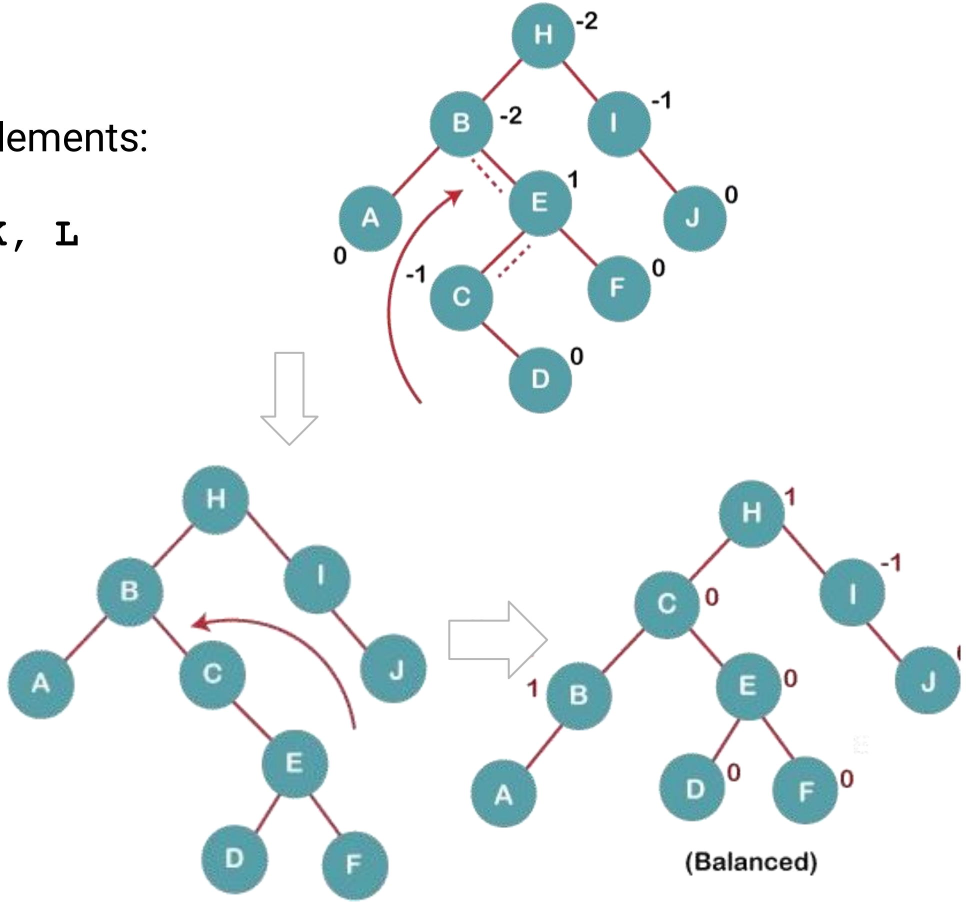
- On inserting E, BST becomes unbalanced as the Balance Factor of I is 2,
- Since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform RL Rotation on node I.
- LR = LL (anti-clockwise) rotation + RR (clockwise)



Insertion in an AVL Tree

Construct an AVL tree with the following elements:

H, I, J, B, A, E, **<u>C, F, D,</u>** **G, K, L**

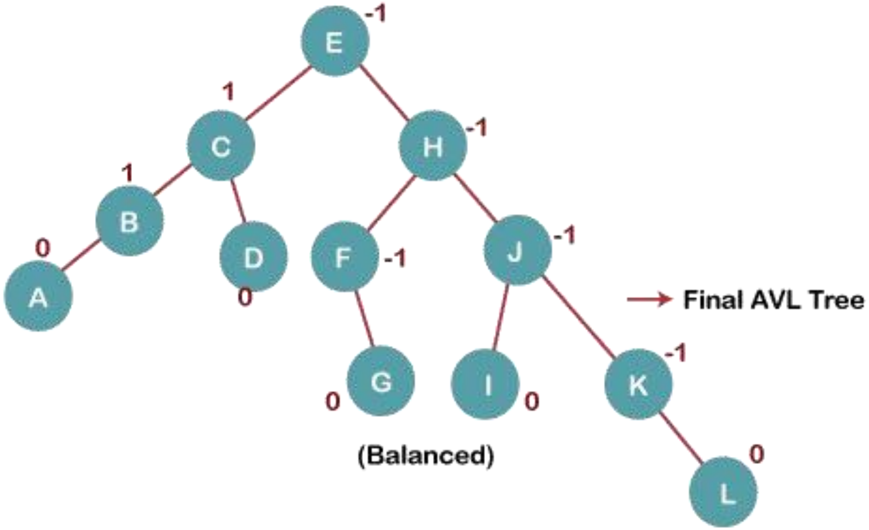


- On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2.
- Since if we travel from D to B we find that it is inserted in the left subtree of right subtree of B, we will perform RL Rotation on node I.
- RL = RR + LL rotation.

Insertion in an AVL Tree

Construct an AVL tree with the following elements:

H, I, J, B, A, E, C, F, D, G, K, L



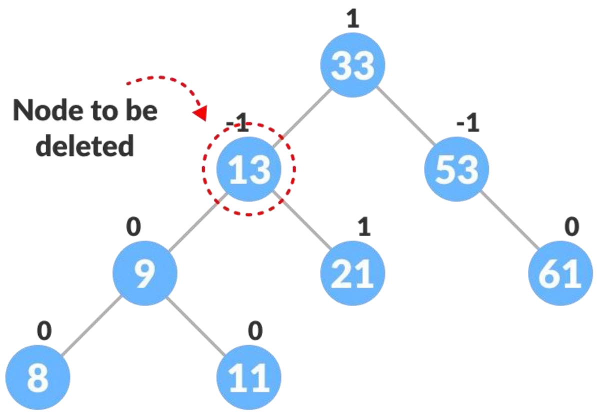
Deletion in an AVL Tree

A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.

Locate `nodeToBeDeleted` (recursion is used to find `nodeToBeDeleted`.

There are three cases for deleting a node:

- If `nodeToBeDeleted` is the leaf node (ie. does not have any child), then remove `nodeToBeDeleted`.
- If `nodeToBeDeleted` has one child, then substitute the contents of `nodeToBeDeleted` with that of the child. Remove the child.
- If `nodeToBeDeleted` has two children, find the inorder successor **`w`** of `nodeToBeDeleted` (i.e., node with a minimum value of key in the right subtree).



ppts:* Deletion in AVL Tree:-

- Here we assume that no repetition is allowed to tree.

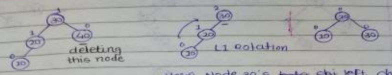
→ Augmenting a data structure:
- bf (balance factor) ને દર વખતે maintain ન કરવી પડે એટલે કે દર વખતે બધાની bf find ન કરવી પડે પરંતુ જેમાં change થયો હોય તેને જ recalculate કરવા માટે Augmenting વપરાય છે (~~for~~ in our case)

ex: Augmented linked list, Augmented queue, Augmented stack etc.

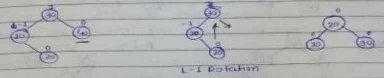
- The added information must be updated and maintained by the ordinary operations on the data structure.
- ~~In~~ when we use Augmenting data structure we check that using this data structure what should be our time and space complexity become.

video:- Rotation for Deletion:-

- After deleting key when node become inbalance we perform a rotation.
- There are 6 types of rotation available.
  1) L1-Rotation
  2) L-1-Rotation
  3) L0 -Rotation
  4) R1 -Rotation
  5) R-1-Rotation
  6) R0 - Rotation

L1:


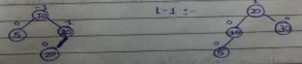
Here, Node 30's ~~bala~~ chi left child balance factor is 1 so it is called L1 Rotation

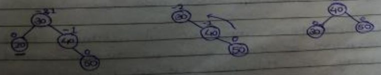
L-1


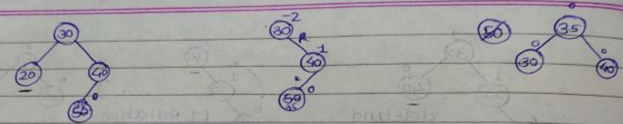
- When we deleting right side we check balanced factor of left child and decide the rotation

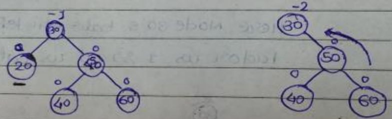
L0

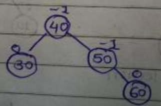

Here we can perform L1 or L-1 rotation.

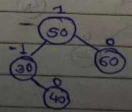
L1:-


R1:-


R1 :-


R0 :-


R1:-


R-1 :-


* Heap

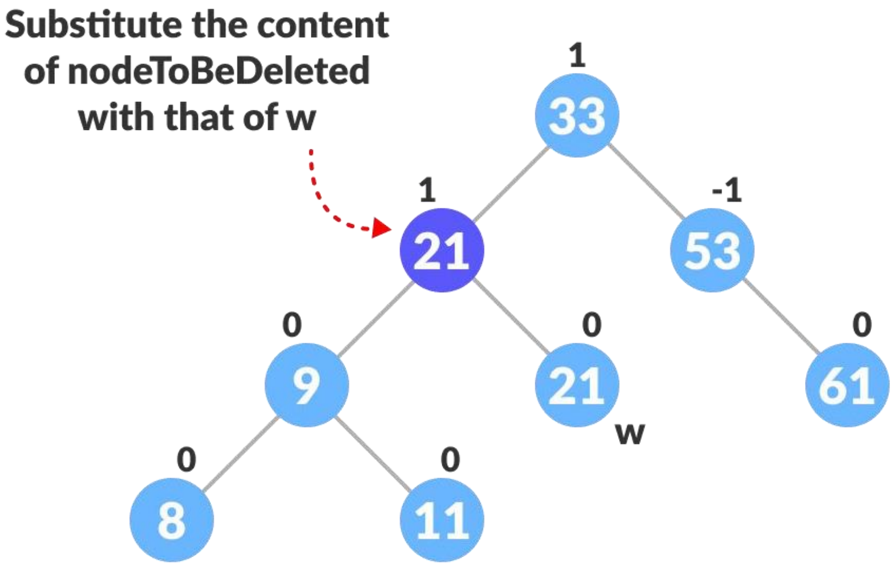
- But bec lis

- hea tre
- He

* In
- I

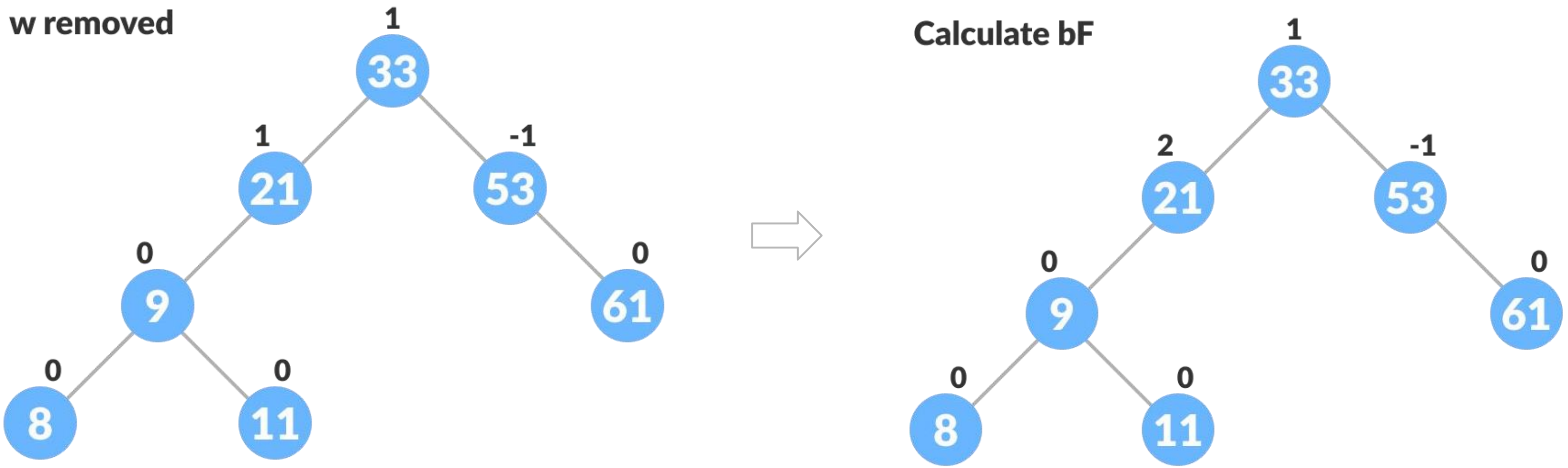
Deletion in an AVL Tree

A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.



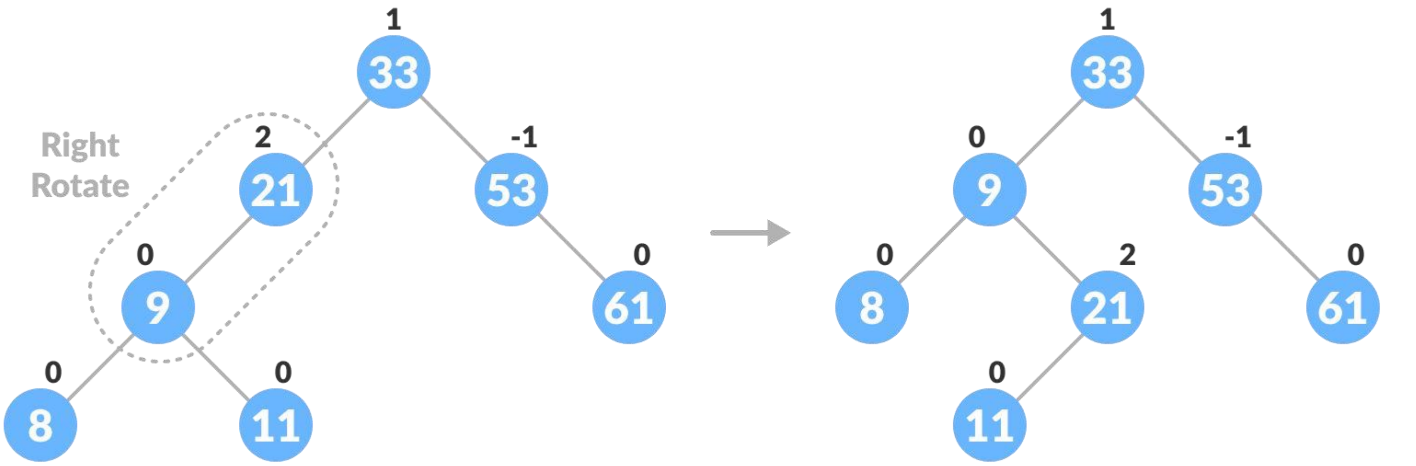
Deletion in an AVL Tree

A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.



Deletion in an AVL Tree

A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.



Augmenting a Data Structure

Augmenting a data structure means storing additional information in it.

- Augmentation is not always straightforward.
- The added information must be updated and maintained by the ordinary operations on the data structure.
- For example,
  - Maintaining the balance factor of each node in the AVL tree

all operation avl1.cpp

Next Lecture

- Heap Tree