Height Balanced Binary Tree

Let us consider the following set of data :

jan, feb, march, apr, may, jun, july, aug, sep, oct, nov, dec.

We know that for a given set of data the Binary Search Tree is not necessarily unique.

In other words, the structure of a Binary Search Tree depends on the ordering of data in the input.

Hence for the above-mentioned data there are 12! Binary search tree possible (each corresponding to an arrangement in the permutation of data).

Next let us define the average search time Γ for an element in a Binary Search as

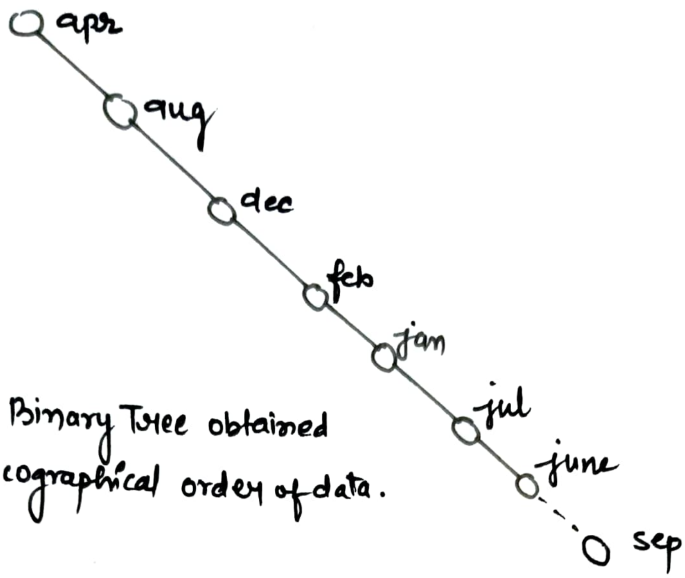
$$\Gamma = \frac{\sum_{i=1}^{n} \tau_i}{n}$$

where

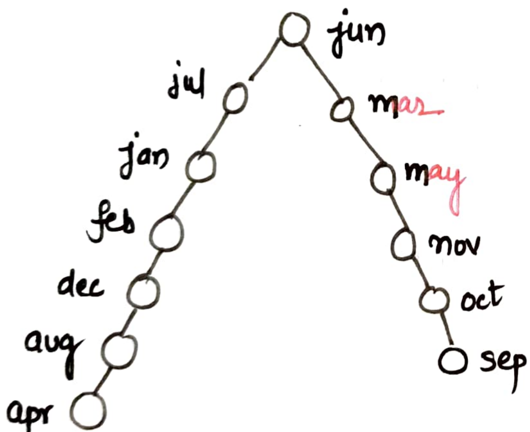
τ_i = no. of comparisons for the i^{th} element.

n = total no. of elements in the Binary Search Time.

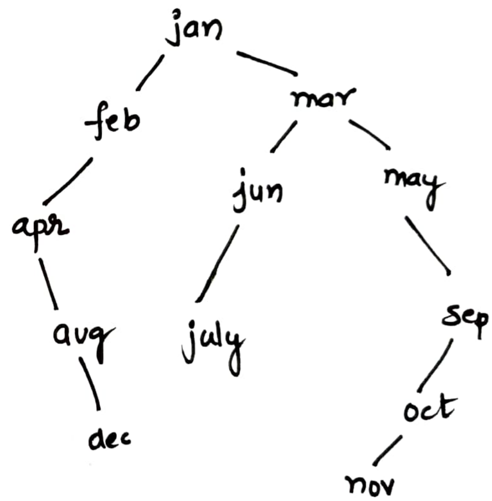
The average search time Γ for the Binary Search Trees



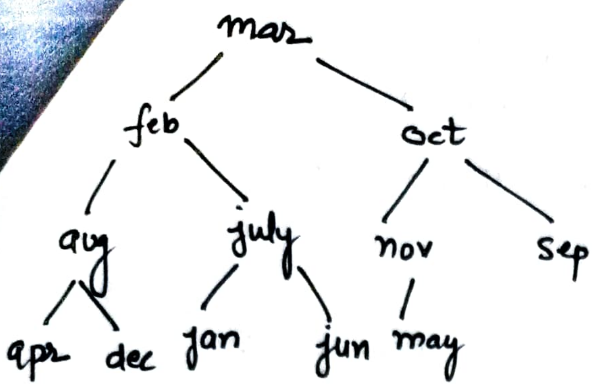
(a) A skewed Binary Tree obtained from lexicographical order of data.



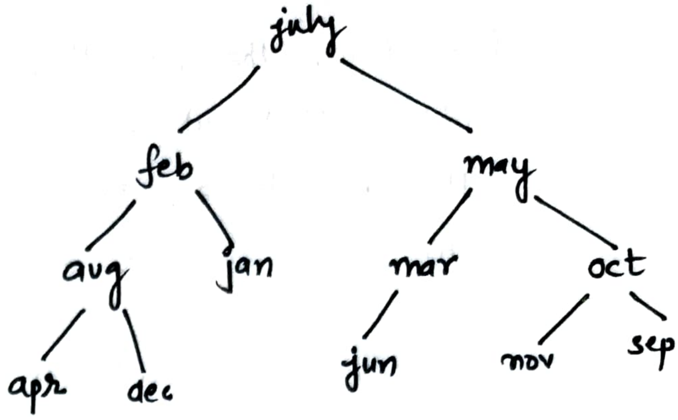
(b) A Binary Search Tree (half skewed version)



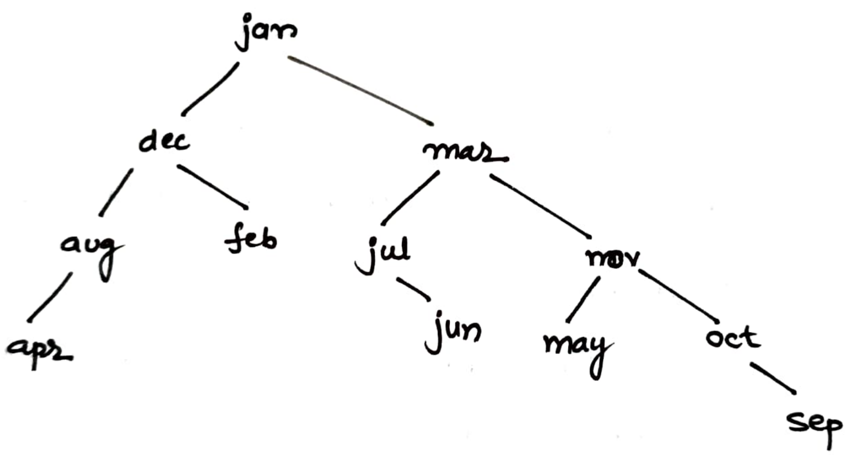
(c) A Binary Search Tree (obtained by Inserting the data into the order of months).



(d) Binary Search Tree in the form of a complete Binary Tree.



(e) A Binary Search tree obtained from a given random ordering of data.



(f) A Binary Search Tree obtained by a special technique

Thus, the average search Time Γ for the Binary Search Trees can be Calculated as

$\Gamma_{(a)} = 6.50$ $\quad$ $\Gamma_{(d)} = 3.08$

$\Gamma_{(b)} = 4.00$ $\quad$ $\Gamma_{(e)} = 3.08$

$\Gamma_{(c)} = 3.50$ $\quad$ $\Gamma_{(f)} = 3.16$

Thus from the preceding calculation, it is evident that out of 6 varieties of representations, the last three representations are efficient from the searching time point of view.

The worst representation is the skewed form of the Binary Search Tree, which needs the highest average search time.

Now the question arises is that for a given set of data how a Binary Search Tree can be constructed so that it will have minimum average search time.

THE ANSWER LIES IN THE CONCEPT OF HEIGHT BALANCED BINARY SEARCH TREE.

A Binary Search Tree can be made by means of calculating the Balance factor of each node.

We first define the term Balance Factor.

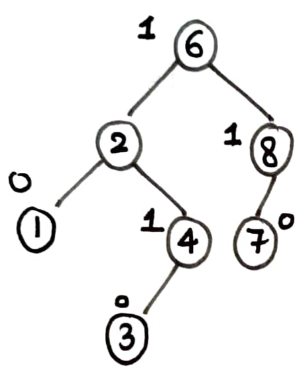
The Balance factor of Binary Tree (bf) is defined as

bf = Height of the left subtree (h_L) - Height of the right subtree (h_R).

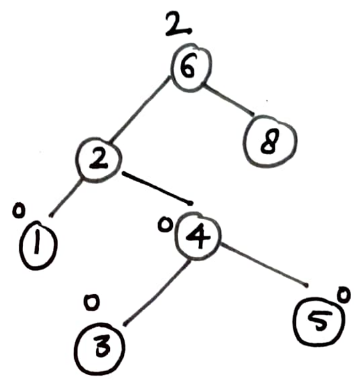
Definition : A Binary Search Tree is said to be height balanced Binary Search Tree if all its node have a Balance factor of 1, 0 or -1. That is,

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

for every node in the tree.



(a) Height Balanced



(b) Height Unbalanced

Two Binary Search Tree with the Balance factor of each node.

It may be noted that a height balanced Binary Tree is always a Binary Search Tree and a complete Binary Search Tree is always height Balance, but the reverse is not true.

The Basic objective of Height Balanced Tree → is to perform searching, insertion, and deletion operation efficiently. These operations may not be with the minimum time but the time involved is less than that of in an unbalanced BINARY SEARCH TREE.

Unbalanced Binary Search Tree can be converted into a height Balanced Binary Tree. Suppose initially there is a height balanced Binary ~~Search~~ Tree. When ever a new node inserted or deleted it may become ~~un~~ unbalanced.

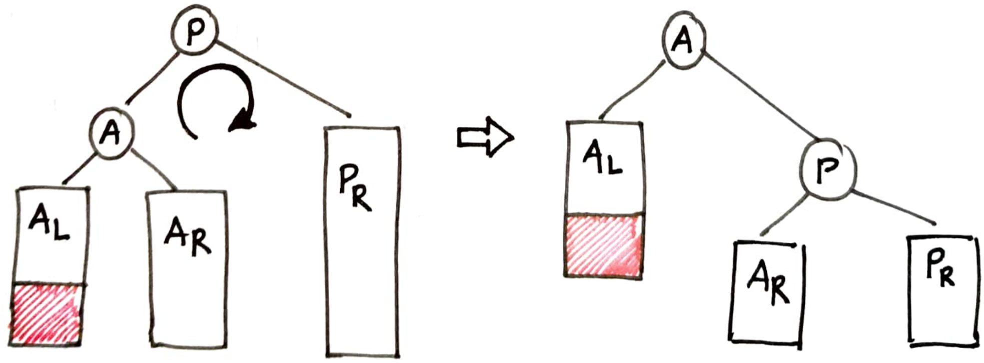
AVL ROTATIONS

In order to balance a tree, an elegant method was devised in 1962 by two Russian mathematicians G.M Adelson-Velskii and E.M Lendis & method k/a AVL Rotation in their honour.

There are four cases of rotations possible which are discussed below:-

Case 1: Unbalance occur due to the insertion in the left sub-Tree of the left child of the pivot node.

THIS CASE IS CALLED LEFT-TO-LEFT INSERTION.



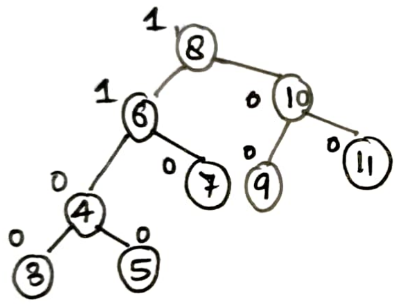
after insertion in the left sub-Tree

after Rotation

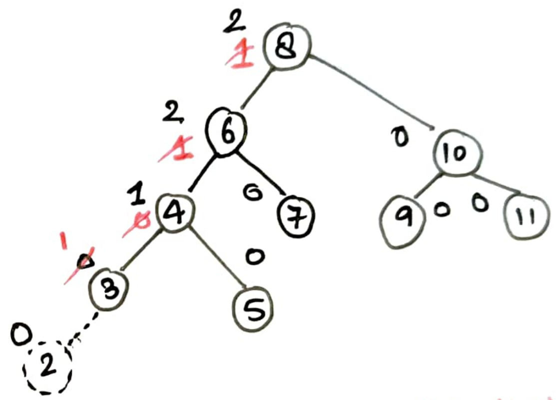
Rotation in the Unbalanced Tree:→

- Right Sub-tree (A_R) of the left child (A) of pivot node (P) becomes the left subtree of P.
- P ~~becom~~ Becomes the right child of A.
- Left Sub-tree (A_L) of A remains the same.

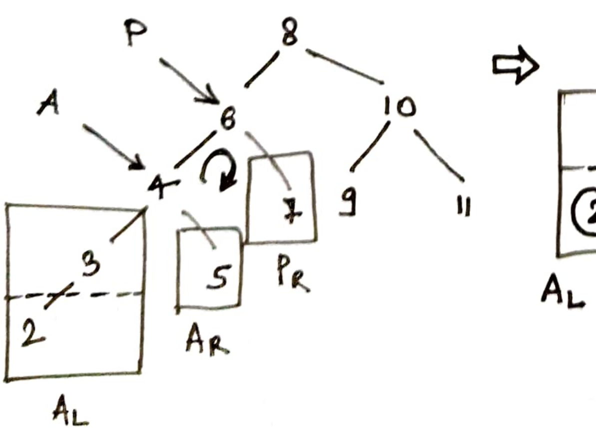
Ex:→



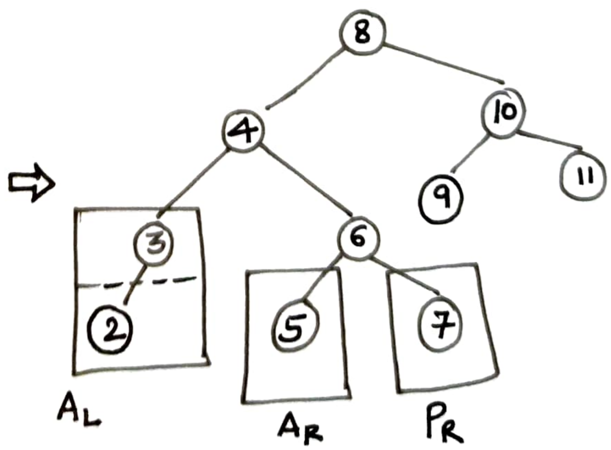
(a) height balanced Binary Tree



(b) After the insertion of 2 into the tree.



(c) AVL Rotation as per the LEFT-To-LEFT



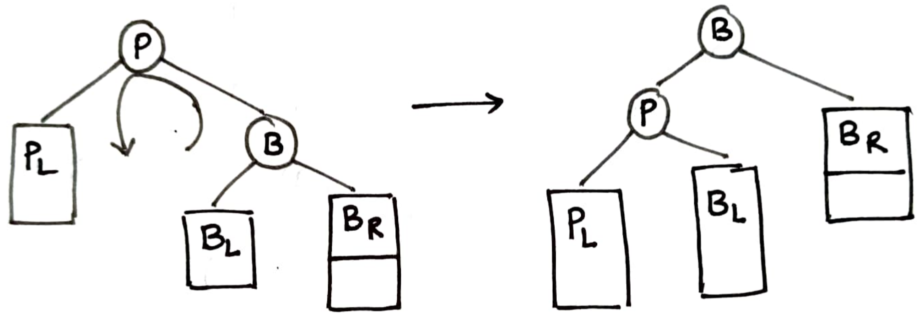
(d) After AVL Rotation.

Case 2: Unbalance occurs due to insertion in the right subtree of the right child of the pivot node.

In this case, the following manipulations in pointers take place.

- Left sub-tree (B_L) of right child (B) of the pivot node (P) becomes the right sub-tree of P.
- P becomes the left child of B.
- Right sub-tree (B_R) of B remains the same.

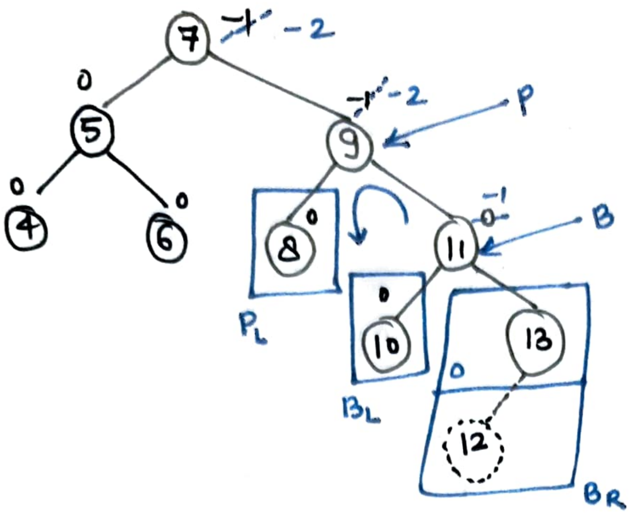
This case is known as RIGHT-TO-RIGHT insertion.



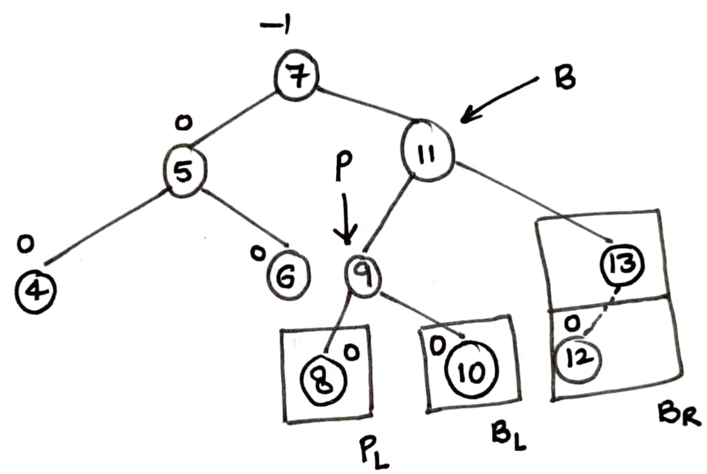
After insertion in the left subtree of the right child of pivot node (P).

After rotation.

AVL Rotation when unbalance occurs due to insertion in the right sub-tree of the right child of the pivot node (RIGHT-TO-RIGHT insertion).



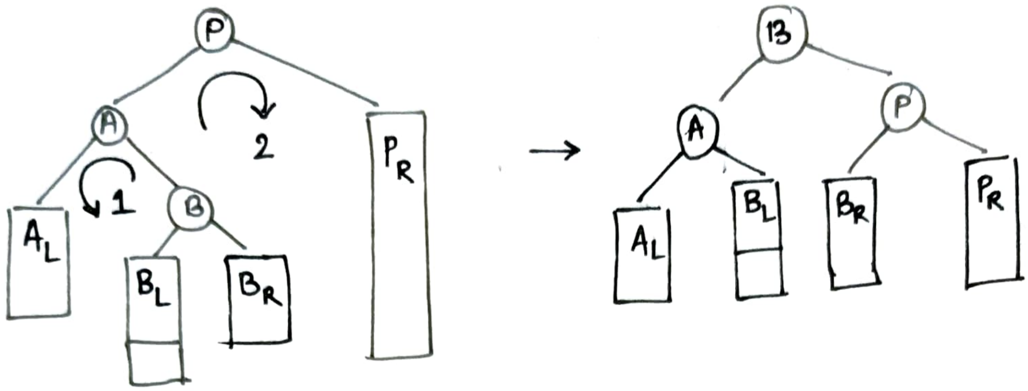
(a) 12 is inserted and this makes the tree unbalanced.



(b) After AVL rotation.

Case 3: Unbalance occurs due to the insertion in the right sub-tree of the left-child of the pivot node.

THIS CASE IS KNOWN AS LEFT-TO-RIGHT Insertion.



After insertion in the right sub-tree of the left child of the pivot node P

After rotation.

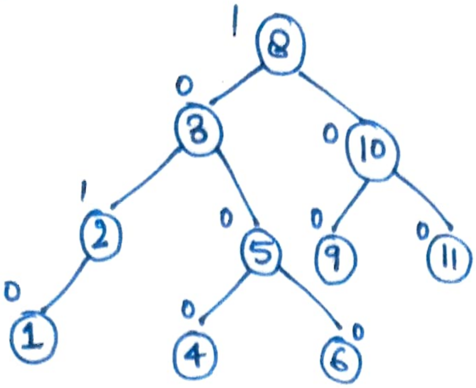
Rotation 1.

- Left sub-tree (B_L) of the right child (B) of the left child of the pivot node (P) becomes the right sub-tree of the left child (A)
- Left child (A) of the pivot node (P) becomes the left child of B.

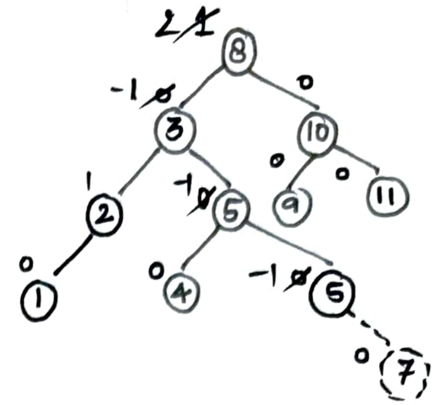
Rotation 2.

- Right sub-tree (B_R) of the right child (B) of the left child (A) of the pivot node (P) becomes the left sub-tree of P.
- P becomes the right child of B.

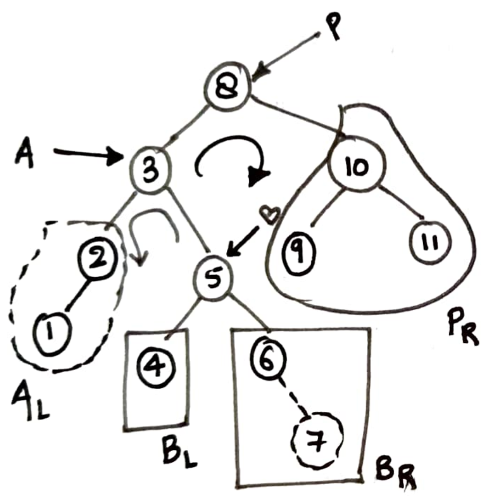
If INSERTION Occurs at B_R instead of B_L, it correspond to Case 3 as well.



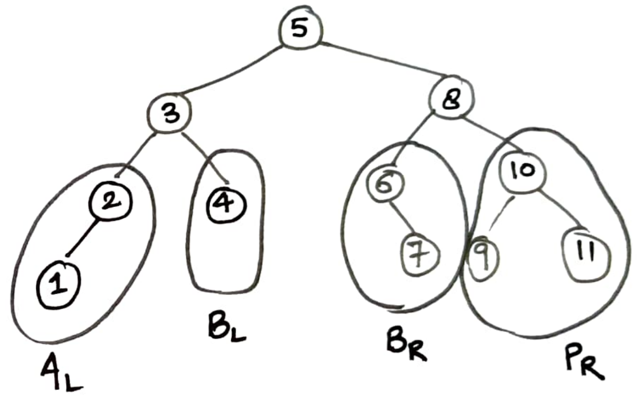
(a) A height Balanced Tree



(b) Insertion of 7 made the tree unbalanced and the balance factors are recomputed.



Node 8 becomes the Pivot node.



(d) After AVL Rotation.

Case 4: Unbalance occurs due to insertion in the left subtree of the right child of the pivot node.

This case is k/a RIGHT-TO-LEFT insertion.

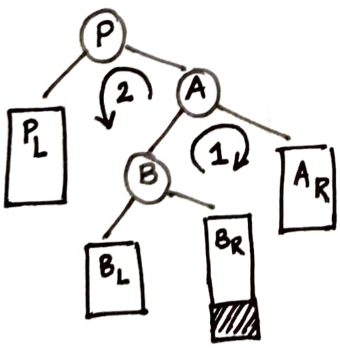
Rotation 1:-

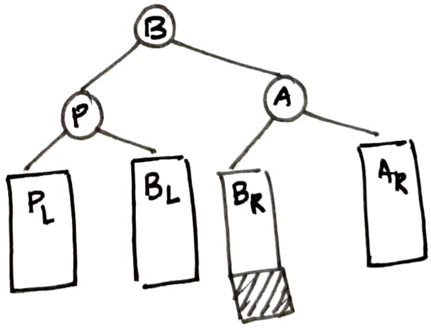
- Right sub-tree (B_R) of the left child (B) of the right child (A) of the pivot node (P) becomes the sub-tree of A
- Right child (A) of the pivot node (P) becomes the right child of B.

Rotation 2:-

- Left sub-tree (B_L) of the right child (B) of the right child (A) of the Pivot node (P) becomes the right sub-tree of P.
- P becomes the left-child of B.

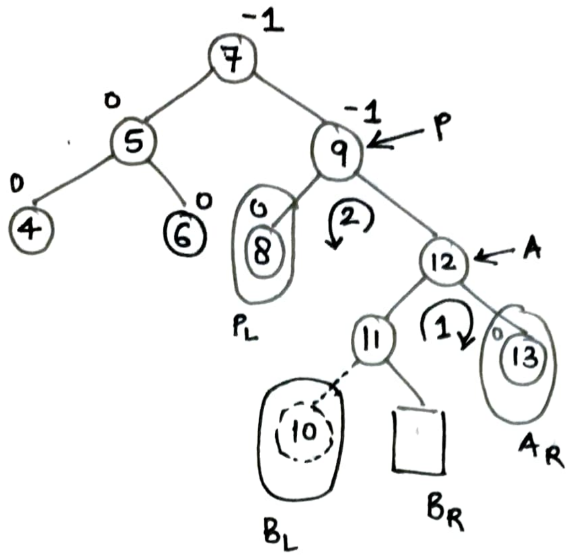
This case is known as RIGHT-To-LEFT insertion.



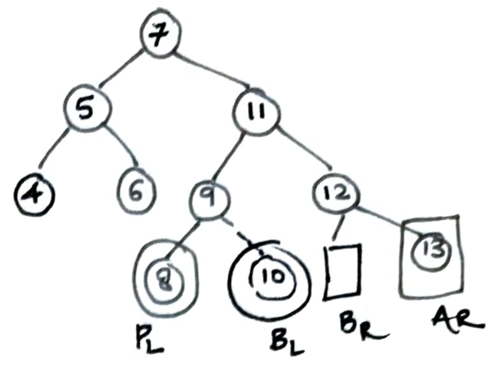


After insertion in the left sub-tree of the right child of the pivot node (P)

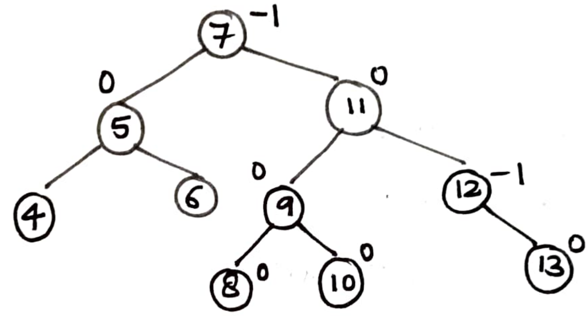
After rotation.



(a) When 10 inserted into the tree



(b) After AVL Rotation



(c) final height balanced tree after AVL rotation.

Implementation for Height Balancing a Tree.

LCHILD	DATA	HEIGHT	RCHILD

Algorithm Compute Height

/* calculate the height of a Binary Tree */

Steps:

1. If (PTR = NULL) then // Height of the empty tree is zero //
2. height = 0
3. Return (height)
4. Else
5. lptr = PTR → LCHILD
6. rptr = PTR → RCHILD
7. h_L = Compute Height (lptr)
8. h_r = Compute Height (rptr)
9. If ($h_L \le h_R$) then // MAximum of left SubTree & Right Sub-Tree
10. height = 1 + h_R
11. Else // $h_L > h_R$
12. height = 1 + h_L
13. EndIf
14. Return (height) // Return height of the tree.
15. EndIf
16. Stop