AVL TREE ALGORITHMS

unbalanced Binary Search Tree can be converted into height balanced BST. Suppose initially there is a height balanced Binary Tree.

Whenever a node is inserted into it (or deleted from it), it may become unbalanced.

<u>following Steps need to be adopted.</u>

1. Insert node into Binary Search Tree :→

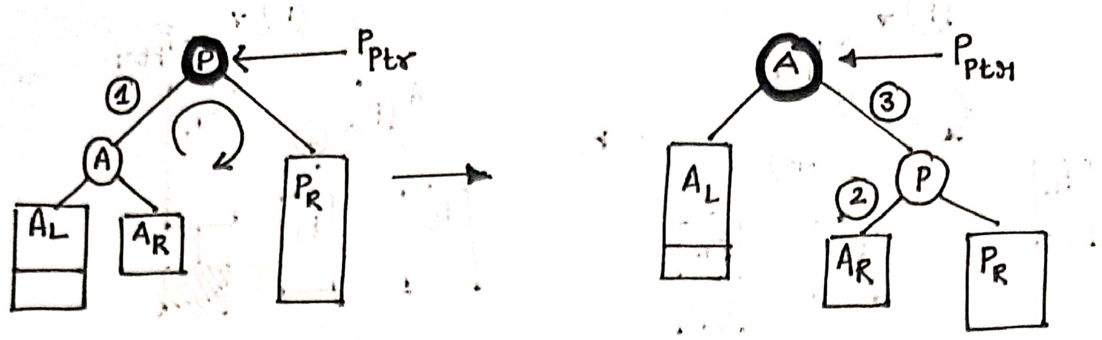
2. Compute the Balance factor :→ On the path starting from the root node to the node newly inserted, compute the balanced factors of each node. It can be verified that a change in Balance Factors will occur only in this path.

3. Decide the pivot node :→ on the path as traced in Step 2, determine whether the absolute value of any node's Balance Factor is switched from 1 to 2. If so, the tree becomes UNBALANCED. The node which has its absolute value of Balance Factor switched from 1 to 2 marked as a Special Node and called the Pivot Node.
   There may be more than one node which has its Balance Factor, |bf| switched from 1 to 2, but the nearest node to the newly inserted node will be the pivot node.

4. Balance the Unbalance Tree :→ It is necessary to manipulate pointers centred at the pivot node to bring the tree back into height balance. This pointer manipulation is well known as AVL Rotation.

Scanned with OKEN Scanner

1. **ALGORITHM LEFT TO LEFT ROTATION**



INPUT: Pointer P_{ptr} to the pivot node

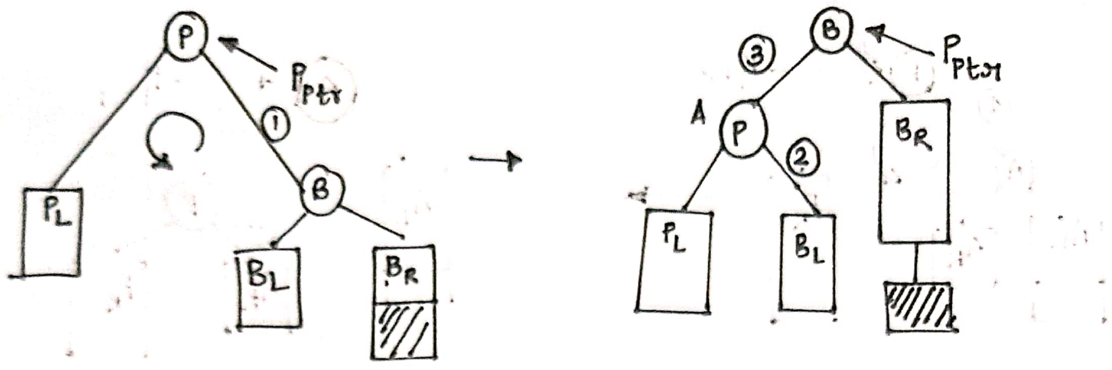
OUTPUT: AVL Rotation corresponding to the unbalance due to insertion in the left sub-tree of the left child of P_{ptr}.

Steps:

1. $A_{ptr} = P_{ptr} \rightarrow$ LCHILD // Pointer initialization as (1)
2. $P_{ptr} \rightarrow$ LCHILD = $A_{ptr} \rightarrow$ RCHILD // Pointer set as (2)
3. $A_{ptr} \rightarrow$ RCHILD = P_{ptr} // Pointer set as (3)
4. $P_{ptr} \rightarrow$ HEIGHT = ComputeHeight(P_{ptr})
5. $A_{ptr} \rightarrow$ HEIGHT = Compute Height (A_{ptr})
6. $P_{ptr} = A_{ptr}$ // Modify the pointer in the parent of pivot.
7. Stop.

Scanned with OKEN Scanner

2. ALGORITHM RIGHT TO RIGHT ROTATION



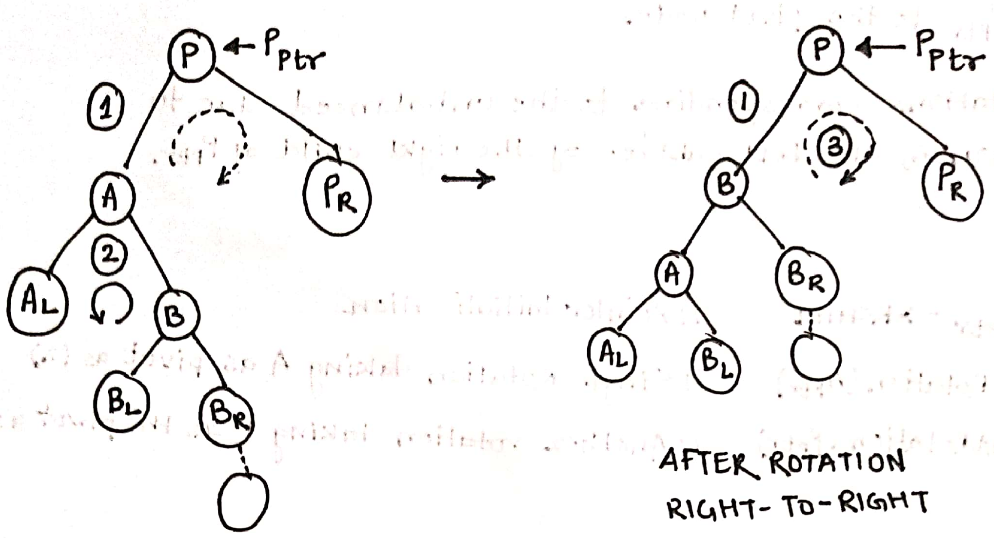
INPUT: Pointer P_{ptr} to the Pivot Node

OUTPUT: AVL rotation corresponding to the unbalanced due to insertion at the right sub-tree of the right child of P_{ptr}.

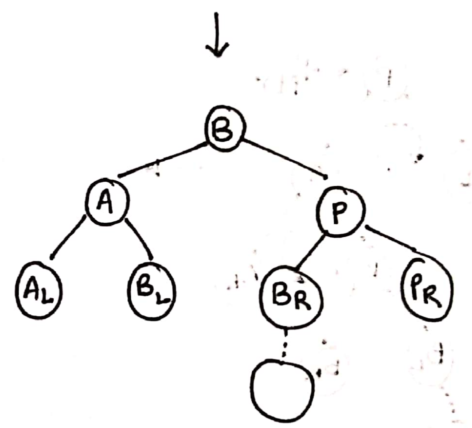
Steps:

1. $B_{ptr} = P_{ptr} \rightarrow$ RCHILD // Pointer Initialization as (1).
2. $P_{ptr} \rightarrow$ RCHILD $= B_{ptr} \rightarrow$ LCHILD // Pointer set as (2)
3. $B_{ptr} \rightarrow$ LCHILD $= P_{ptr}$ // Pointer set as (3)
4. $P_{ptr} \rightarrow$ HEIGHT = computeHeight(P_{ptr}) // Recompute the height of P and B.
5. $B_{ptr} \rightarrow$ HEIGHT = computeHeight(B_{ptr})
6. $P_{ptr} = B_{ptr}$ // Modify the pointer field in the parent of pivot node.
7. Stop.

3. ALGORITHM LEFT TO RIGHT ROTATION



AFTER ROTATION
RIGHT-TO-RIGHT



AFTER ROTATION
LEFT-TO-LEFT

INPUT: Pointer P_{ptr} to the pivot node.

OUTPUT: AVL Rotation corresponding to the unbalance due to Insertion in the right sub-tree of the left child of P_{ptr}.

Steps:

1. $A_{ptr} = P_{ptr} \rightarrow$ LCHILD // Pointer Initialization as (1)
2. RightToRightRotation (A_{ptr}) // single rotation taking A as pivot as (2)
3. LeftToLeftRotation (P_{ptr}) // Another rotation taking P as pivot as (3)
4. Stop

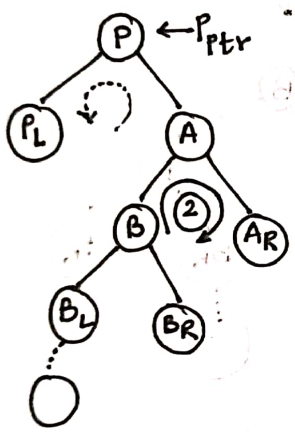
4. ALGORITHM RIGHT TO LEFT ROTATION

INPUT: Pointer P_{ptr} to the pivot node.

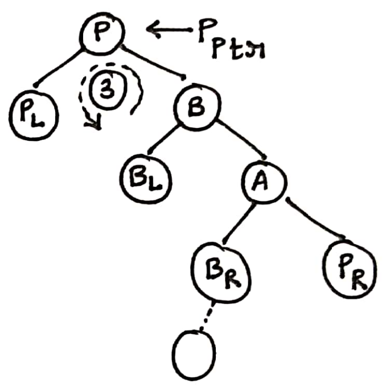
OUTPUT: AVL Rotations corresponding to the unbalanced due to insertion in the left subtree of the right child of P_{ptr}.

Steps:

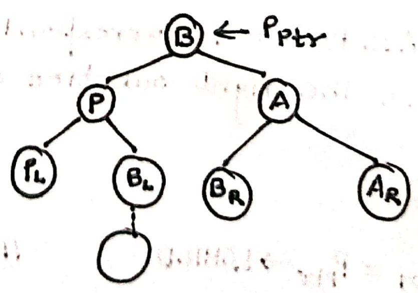
1. $A_{ptr} = P_{ptr} \rightarrow$ RCHILD // Pointer Initialization
2. LeftToLeftRotation (A_{ptr}) // Single Rotation taking A as pivot as (2)
3. RightToRightRotation (P_{ptr}) // Another rotation taking P as the pivot as (3)
4. Stop



AFTER INSERTION



AFTER ROTATION
LEFT-To-LEFT



AFTER ROTATION
RIGHT-To-RIGHT

ALGORITHM InsertHBT

Steps:

```
1.  ptr=ROOT
2.  If (ptr= NULL) then
3.       ptr = Nptr
4.       ptr -> HEIGHT=1
5.       Return( )
6.  Else
7.     If (Nptr -> DATA < ptr -> DATA) then
8.          InsertHBT (ptr-> LCHILD, Nptr)
9.          Lptr = ptr-> LCHILD
10.         Rptr = ptr-> RCHILD
11.         If (Rptr = NULL) then
12.              hR=0
13.         Else
14.              hR = Rptr ->HEIGHT
15.              hL = Lptr-> HEIGHT
16.              bf = hL-hR
17.              If (bf=2) then
18.                 If (Nptr-> DATA < Lptr->DATA) then
19.                     LefttoLeftRotation (ptr)
20.                 Else
21.                     LefttoRightRotation (ptr)
22.                 EndIf
23.              ptr-> HEIGHT = ComputeHeight(ptr)
24.           EndIf
25.        EndIf
```

```
26. Else
27.     If (Nptr->DATA > ptr->DATA) then
28.         InsertHBT (ptr->RCHILD, Nptr)
29.         Rptr = ptr->RCHILD
30.         Lptr = ptr->LCHILD
31.         If (Lptr = NULL) then
32.             hL = 0
33.         Else
34.             hL = Lptr->HEIGHT
35.             hR = Rptr->HEIGHT
36.             bf = hL - hR
37.             If (bf = -2) then
38.                 If (Nptr->DATA > Rptr->DATA) then
39.                     RIGHT TO RIGHT ROTATION (ptr)
40.                 Else
41.                     RIGHT TO LEFT ROTATION (ptr)
42.                 Endif
43.                 ptr->HEIGHT = computeHeight (ptr)
44.             Endif
45.         Endif
46.     Else
47.         print Nptr->DATA "is already exist in the tree"
48.     Endif
49. Endif
50. Endif
51. Stop.
```