Assignment No. 8

Title :- Implement height balanced tree.

Aim :- To learn & implement AVL tree with different operation.

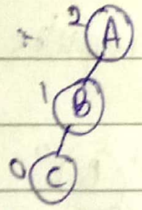
Problem statement :- A dictionary stores keywords & its meaning provide the facility for adding new keyword deleting keyword, updating value for an entry. provide facility to display data in ascending & descending order. also find how many max comparision required for finding any keyword. use height balanced tree & find complexity for finding keyword.

Objective :- To learn the basic concepts regarding self-balancing tree & its need.
To perform different operation on AVL tree.

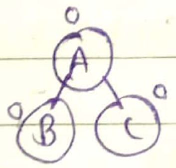
Theory :-

Height balanced tree :- Height balance tree is data structure to maintain the height of balance factor (0, 1, -1)
AVL tree checks height of left & Right sub-tree.

e.g :-



Not Balanced



balanced

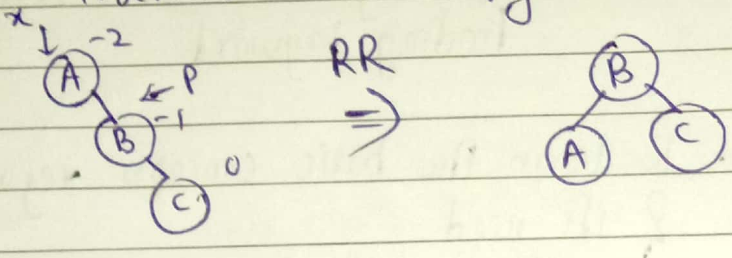
AVL Rotations :

To balance itself, an AVL tree may perform four rotations.

i) Left rotation
ii) Right rotation
iii) Left Right rotation
iv) Right Left rotation.

i) Right rotation : if tree becomes unbalanced when a node is inserted into the right subtree of right subtree.

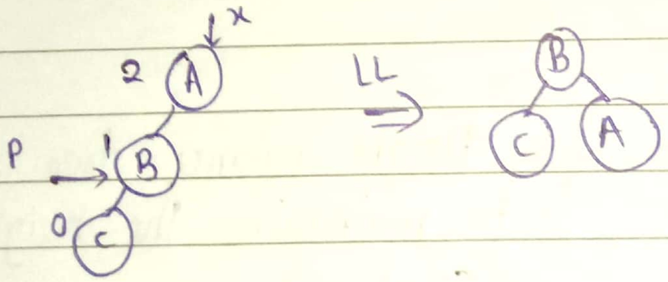
e.g.



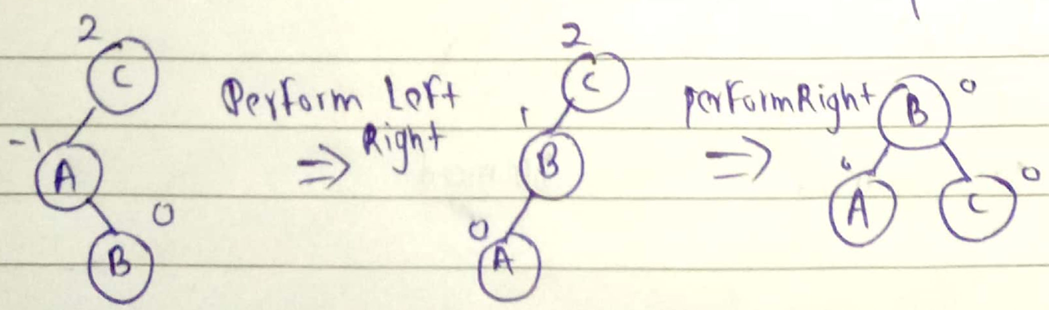
P beomes parent of x & x become left child of P.

ii) Left Rotation : if tree become unbalanced when a node is inserted into left subtree.

e.g.

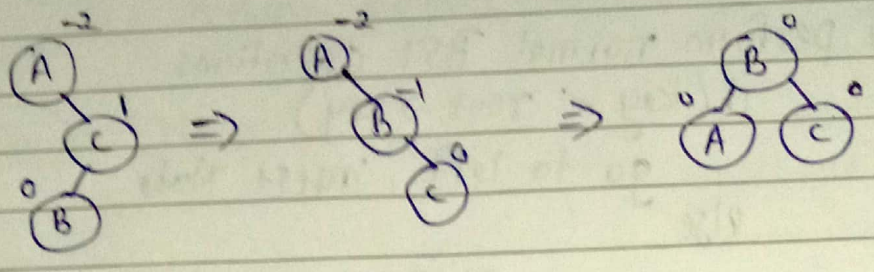


iii) Left-Right rotation : A Node has been inserted into the right subtree & left subtree. This makes tree unbalaned & we require LR rotation.



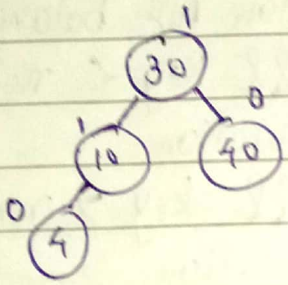
iv) Right - Left Rotation : A Node has been inserted into left subtree of right subtree this make tree unbalanced.

e.g.



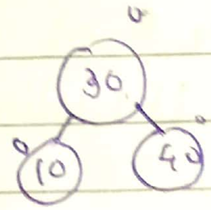
• Operations :-

i) Insertion : while inserting a node, we calculate the balance factor of tree & make it balanced.



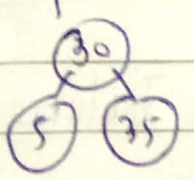
ii) deletion :- After deleting a node, we have to recalculate the balance factor of tree to maintain tree balance

delete 4 :

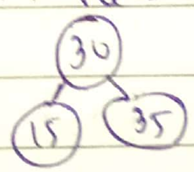


Balanced.

iii) Updation : Updating involve searching the particular node & replace old value with new value.



old = 5
new = 15



Algorithms :-

• Insertion :-

Step 1.) perform normal BST Operations
if(key < root -> key)
go to left, insert Node
else
go to Right insert Node

Step 2.) Update the height of ancestor node & check whether balance factor is balanced or not.
if (Not balanced)
check the condition for balance factor
if(balance >1 && key < node->left->key)
perform Right rotation
if(balance< -1 && key > node-> right -> key)
perform Left rotation
if(balance<-1 && key < node-> right->key)
perform RL rotation
if(balance >1 && key > node -> left -> key)
perform LR rotation

Step 3.) END

Getting Balance factor :-

Step 1.) Read the node which Balance factor you want

Step 2:) if Node is Null
return 0
else
return height(node -> left) - height(node -> Right)

Step 3:) END

- Getting height of tree.

Step 1:) if root is null then return 0.
step 2:) if root -> left is null
initialize left height to 0
Step 3:) if root -> Right is null
initialize Right height to 0.
Step 4:) if root -> Right is not null
increment Right height = 1 + root -> right -> height
Step 5:) if root -> left is not null
increment left height = 1 + root -> left -> height
Step 6:) check (left height > Right height) is true
return left height
else
return Right height
Step 7:) END

Rotation algorithms :-

i) Right rotate :-

step 1) Read the node which you want to rotate say *node y
step 2) Let x be the left of y & z be the right of x
step 3) perform the rotation S.T. x becomes parent of y
z becomes left child of y.
$x \to right = y$
$y \to left = z$
step 4:) Update the height of x & y
step 5:) return x;

ii) LeftRotate :-

Step 1:) Read the node which you want to rotate say *node
Step 2:) Let x be the right of y & z be the left of x
Step 3:) perform rotation S.T. x becomes parent of y
& z becomes right child of y
$x \to left = y$
$y \to Right = z$
step 4:) update the height of x & y
step 5:) Return x.

Deletion :-

step 1:) perform standard BST delete for w.
step 2:) starting from w, travel up & finish find

unbalanced node. let z be the first unbalanced node, y be the larger height child of z & x be the larger height child of y.

Step3:) Re-balance the tree by performing appropriate rotation. with rooted with z there might of 4 cases

a) Left Left case
b) Left Right case
c) Right Right case
d) Right left case.

Step 4:) END

• Update:
Step 1:) Create new node *temp
Step 2:) assign root to temp.
step 3:) if (root == Null) then
return null
else
check (the temp of key is equal to key is true)
then update meaning
else
make recursive call with root → left & key with root → Right & key.

Step 4:) Return temp
step 5:) Stop.

- Ascending order :-

step 1.) if root is Null print Null

step 2.) if root is not null

make recursive call with root → right

print → data

make recursive call with root → left

step 3.) stop.

- Application & time complexity :-

Searching - O(log n) worst case

Insertion - O(log n) worst case

Deletion - O(log n) worst case.

i) If application involved frequent insertion & deletion, AVL is not as much efficient, so where insertion & deletion less frequent AVL should be prefered.

ii) For lookup intensive application, avl tree are much preferred than red-black tree as they are strictly balanced.

- Conclusion :-

i) BST can't control over order in which data comes for insertion Due to this BST sometimes become skewed. Since AVL tree are strictly-balanced tree they can be used for more faster application. than BST.

ii) we have implemented AVL tree in data structure.