

The **height** of a node is the length of the longest downward path to a leaf from that node. The height of the root is the **height of the tree**.

The **depth** of a node is the length of the path to its root (i.e., its root path). This is commonly needed in the manipulation of the various self-balancing trees, AVL Trees in particular. The root node has depth zero, leaf nodes have height zero, and a tree with only a single node (hence both a root and leaf) has depth and height zero. Conventionally, an empty tree (tree with no nodes, if such are allowed) has depth and height −1.

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |



struct BinaryTree {

int item;

int height;

BinaryTree \*left;

BinaryTree \*right;

};

|  |  |
| --- | --- |
| Node address | |
| \*left | Item | | height | | \*right |

|  |  |
| --- | --- |
| Node address | |
| \*left | Item | | height | | \*right |

|  |  |
| --- | --- |
| Node address | |
| \*left | Item | | height | | \*right |

An AVL tree (Georgy Adelson-Velsky and Landis' tree, named after the inventors) is a [self-balancing binary search tree](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree). In an AVL tree, the [heights](http://en.wikipedia.org/wiki/Tree_height) of the two [child](http://en.wikipedia.org/wiki/Child_node) subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. The benefit of AVL trees over Binary Search Trees is that the number of comparisons required, i.e. the AVL tree's height, is guaranteed never to exceed log(n).

An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

Trees are balanced through rotations. Tree rotation is an operation on a [binary tree](http://en.wikipedia.org/wiki/Binary_tree) that changes the structure without interfering with the order of the elements. A tree rotation moves one node up in the tree and one node down. It is used to change the shape of the tree, and in particular to decrease its height by moving smaller subtrees down and larger subtrees up, resulting in improved performance of many tree operations. When a subtree is rotated, the subtree side upon which it is rotated increases its height by one node while the other subtree decreases its height. This makes tree rotations useful for rebalancing a tree

If positive number, it is left heavy, if negative number it is right heavy.

if balance >1 or balance <-1 then rotate.

If >1 go the left child (left heavy)

If child is negative (right heavy)

(right heavy, inside heavy) double rotate(Left rotate, right rotate)

else (left heavy)

right rotate

If <-1 go to the right child (right heavy)

If child is positive, (left heavy)

(left heavy, inside heavy) double rotate (right rotate, left rotate)

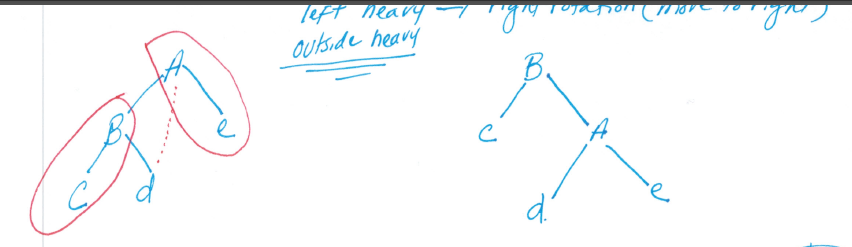
else

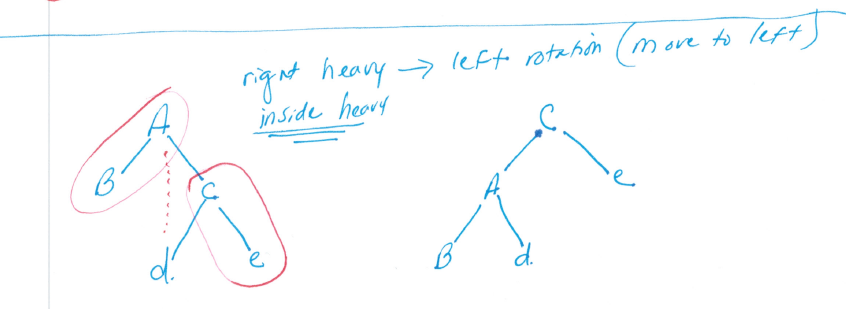
left rotate (right heavy)

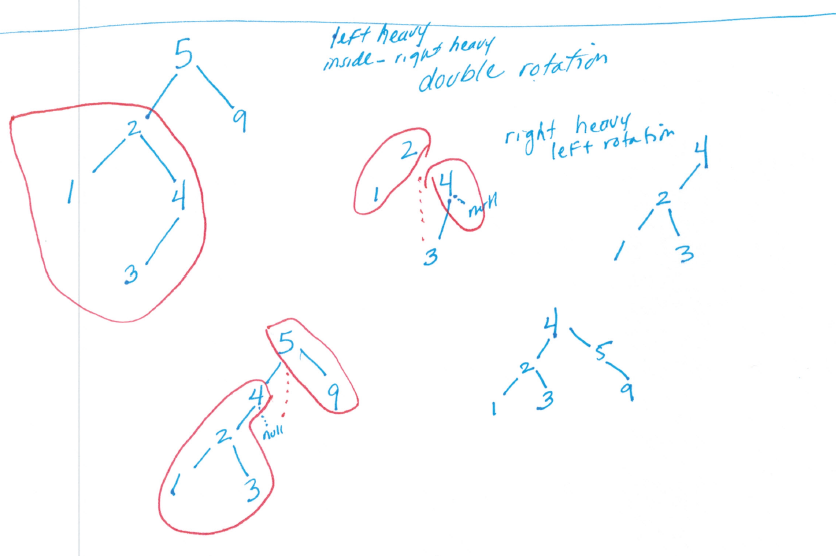
If outside heavy on the left, rotate to the right (balance is positive)

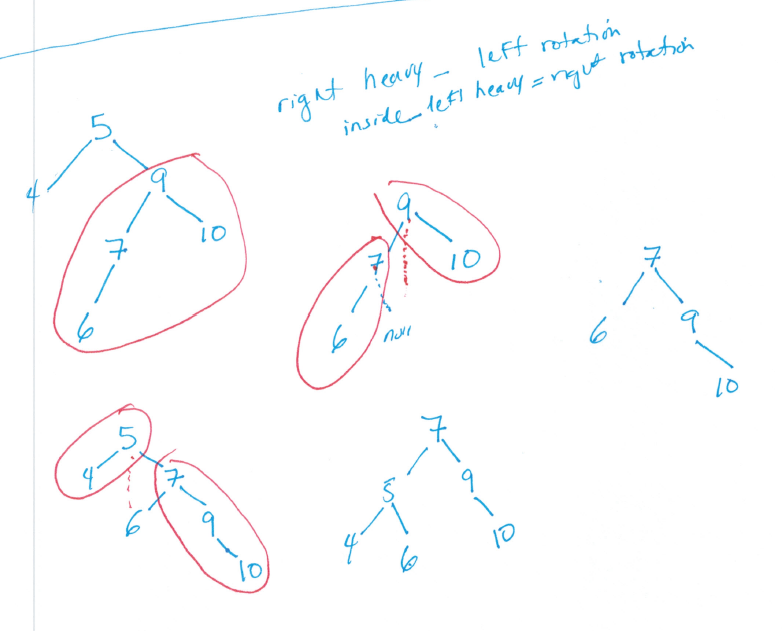
If outside heavy on the right, rotate to the left (balance is negative)

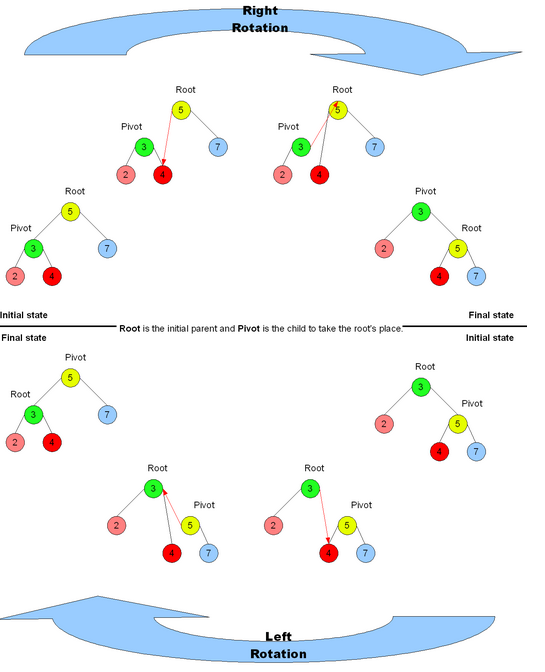
|  |  |
| --- | --- |
| Left heavy  (Outside)  Right heavy  (outside)  Left heavy  Right heavy  (inside)  Right heavy  Left heavy  (inside) |  |











//This is not a recursive algorithm

struct node \* rebalance(struct node \*node){

node->height = max(node->left->height, node->right->height) + 1;

int balance = getBalance(node); //node->left - node->right

/\*

rotation conditions based on balance

node right heavy <-1

right child is left heavy >0

//double rotation right, left case #3

node->right=rotate right on child

return rotate left on node

else

return rotate left on node //single left rotation case #2

node is left heavy> 1

left child is right heavy < 0

//double rotation left, right case #4

node->left=rotate left on child

return rotate right on node

else

return rotate right on node //single right rotation case #1

(The first arg in parenthesis deals with the node whose balance is > 1 or < -1)

(The second arg in parenthesis deals with the node under it and its balance)

(+,+) // 1st is left heavy, 2nd it is outside heavy (single right rotation

(-,-) // 1st is right heavy, 2nd is outside heavy (single left rotation)

(+,-) // 1st is left heavy, 2nd is inside heavy (double rotation, left and right)

(-,+) // 1st is right heavy, 2nd is inside heavy(double rotation, right, and left)

The numbers in paranthesis are the balance factors

case 1 ============== single right rotation + , +

5 (2) 3

3(1) 1 5

1

case 2 =============== left rotation - , -

5(-2) 6

6 (-1) 5 7

7

case 3 =============== right (inside), left (outside) rotation (Double)

5 (-2) 7 7

8 (1) 8 5 8

7

case 4 ============= left (inside), right (outside) rotation (Double)

5(2) 3 3

2(-1) 2 2 5

3

\*/

}

//non-tail recursive algorithm because of rebalance

struct node\* insert(struct node\* node, int key)

{

//recursive Code for inserting a node

//When insert happens set height to 0 for the node

if (node == NULL)

return(newNode(key));

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

node=rebalance(node); //update heights and rebalance

return node;

}

struct node \*leftRotate(struct node \*x){

struct node \*y=x->right;

//add more code to rotate to the left, update heights for x and y

//return root of the tree

}

struct node \*rightRotate(struct node \*x){

struct node \*y=x->left;

//add more code to rotate to the right, update heights for x and y

//return root of the tree

}